SIMD Within A Register on Linear Feedback Shift Registers

A Project

By Karl Ott

Presented to the faculty of University of Alaska Fairbanks
In Partial Fulfillment of the Requirements of
MASTER OF SCIENCE IN COMPUTER SCIENCE

Fairbanks Alaska
April 22, 2015

SIMD Within A Register on Linear Feedback Shift Registers

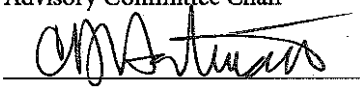
A Project

By Karl Ott

RECOMMENDED:

2015-04-21

Advisory Committee Chair — Date



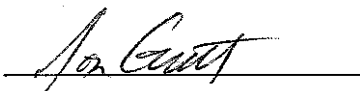
4/21/15

Advisory Committee Member — Date

4/21/15

Advisory Committee Member — Date

APPROVED:



4/21/15

Department Head, Computer Science Department — Date

Dean, College of Engineering and Mines — Date

Dean of the Graduate School — Date

Date

Abstract

Linear feedback shift registers (LFSRs) are used throughout a subset of cryptography. They have long been deployed as a means to generate a pseudo-random number stream. The random number generation provided by the LFSRs has been utilized in stream ciphers ranging from consumer to military grade. For example GSM privacy relies on the A5/1 stream cipher which in turn relies on LFSRs to generate the keystream. They are deployed because they are easy to construct, yet still provide strong cryptographic properties. The scope of this project is to speed up the simulation of LFSRs. The method of speeding up LFSRs is to use parallel operations to operate on multiple LFSRs at once. This is accomplished by using a method of SIMD. The method is SIMD within a register (SWAR). SWAR uses general purpose machine registers (eg. rax on an x86_64 machine). This means that 64 LFSRs can be simulated at once with one machine register using SWAR. This has the trade off of latency vs throughput.

Acknowledgments

Firstly I would like to extend my sincere gratitude to my advisor, Dr. Lawlor. He has been indispensable through the course of this project by serving as a constant source of good ideas, invaluable advice, quality feedback, and excellent instruction as a professor. One would be hard pressed to find a better advisor.

I would also like to thank the other members of my committee, Dr. Genetti and Dr. Hartman. Both have taught me a significant amount as professors during my undergraduate career and continue to do through gradate school. This project would not have been possible without the education provided by Dr. Hay, Dr. Chappell, and Dr. Nance. I would also like to thank the rest of the professors who have taught me through my academic career.

Fellow graduate students Matt VanVeldhuizen and Ben Bettisworth deserve recognition for putting up with my constant barrage of "sanity checks". Drew "Don/Juan" Lanning and Chris Woodruff also deserve thanks for being such excellent friends over the course of my life thus far.

Lastly, I would like to than my family. Specifically my parents for instilling me with a thirst for knowledge, and doing anything and everything to put me on a path towards success. Their constant support has been of the utmost importance throughout my academic career.

Table Of Contents

List of Figures

List of Tables

1 Introduction

SIMD within a register is provides significant speedups when simulating linear feedback shift registers. Simulation of linear feedback shift register (LFSRs) in software is straight forward with a serial implementation. For instance a 64 bit length LFSR can be represented in C like so: uint64_t lfsr. Unfortunately it also lacks any sort of improvements with parallel operations. However, a modification to the layout of the LFSRs allows for parallel operations. This can be accomplished by using a single instruction multiple data (SIMD) within a register (SWAR) method. The LFSRs can be stacked and grouped vertically. This means that one operation can operate on many LFSRs in parallel. However, this incurs a trade off of latency vs throughput. That is the overall time to execute will increase but for more LFSRs simulations, thus an increase in throughput. As the LFSRs are stacked vertically this can be represented by an array. An example in C would look like uint64_t lfsrs[64]. This gives 64 LFSRs, based on the width of the array type, which are 64 in length determined by the length of the array.

2 Background

2.1 Linear Feedback Shift Registers

A linear feedback shift register (LFSR) is a shift register that has an input bit that is determined from a linear function of its previous state. Generally the most common linear function used is exclusive-or (XOR). Meaning that a LFSR is commonly a shift register with its input driven by the XORed result of some of the bits of value in the shift register. Often LFSRs are used as pseudo-random number generators because of their simple construction. For instance, LFSRs can be constructed with flip-flops and discrete logic gates, most likely XOR. They offer long periods and a uniformly distributed output stream. The pseudo-random number streams can be utilized in stream ciphers like the A5/1 cipher which provides privacy in the GSM cellular network.

The ease of construction also can be seen when a LFSR is simulated in software. The code below shows an example of a LFSR that is 4 bits long and the fed back result comes from the 2nd and 3rd bit in the shift register.

```
uint8_t lfsr, bit, i;
lfsr = 0x08;
for(i = 0; i < 128; i++) {
    //get tap output
    bit = ((lfsr >> 3) ^ (lfsr >> 2)) & 0x1;
    //shift register
    lfsr = (lfsr << 1) | bit;
    lfsr &= 0x0f;
}
```

Above the `lfsr` holds the state of the LFSR and `bit` holds the feedback value. The for loop is how the LFSR gets clocked, with `>>` and `<<` meaning right shift and left shift respectively. The linear function used here is XOR which is denoted by `^`, while `&` represents AND and `|` OR.

2.2 A5/1

The A5/1 cipher is the standard cipher that provides voice privacy in the GSM network. The cipher utilizes three LFSRs and an irregular clock. The irregular clock is used to facilitate resistance against cryptanalysis [2]. The LFSRs are clocked with a majority rule. Meaning that the majority of the LFSRs dictate which ones are clocked. The three LFSRs are varying in length and tapped and clocked at different bits [7]. These values are shown in Table 1.

LFSR number	Length	Clocking bit	Tapped bits
1	19	8	13, 16, 17, 18
2	22	10	20, 21
3	23	10	7, 20, 21, 22

Table 1: Specifications of LFSRs in A5/1 [3]

The operation of the cipher can be decomposed into two main functions. A setup function and a generate keystream function. The setup function starts with a set of zeroed LFSRs. A 64 bit key is

generated on the mobile device's SIM card and a challenge is issued to service provider to share the 64 bit key. The key is then XORed in to the oth bit of the LFSRs. The LFSRs are then all clocked and the cycle continues until the key is consumed. After the key is consumed into the LFSRs a publicly known 22 bit frame counter is XORed in the same manner. Once both the key and the frame are consumed the LFSRs are majority clocked 100 times with output discarded. The LFSRs are now ready to generate two 114 bit keystreams. One keystream is used for upstream and the other for downstream [4]. A diagram of the structure of the A5/1 LFSRs is shown in Figure 1.

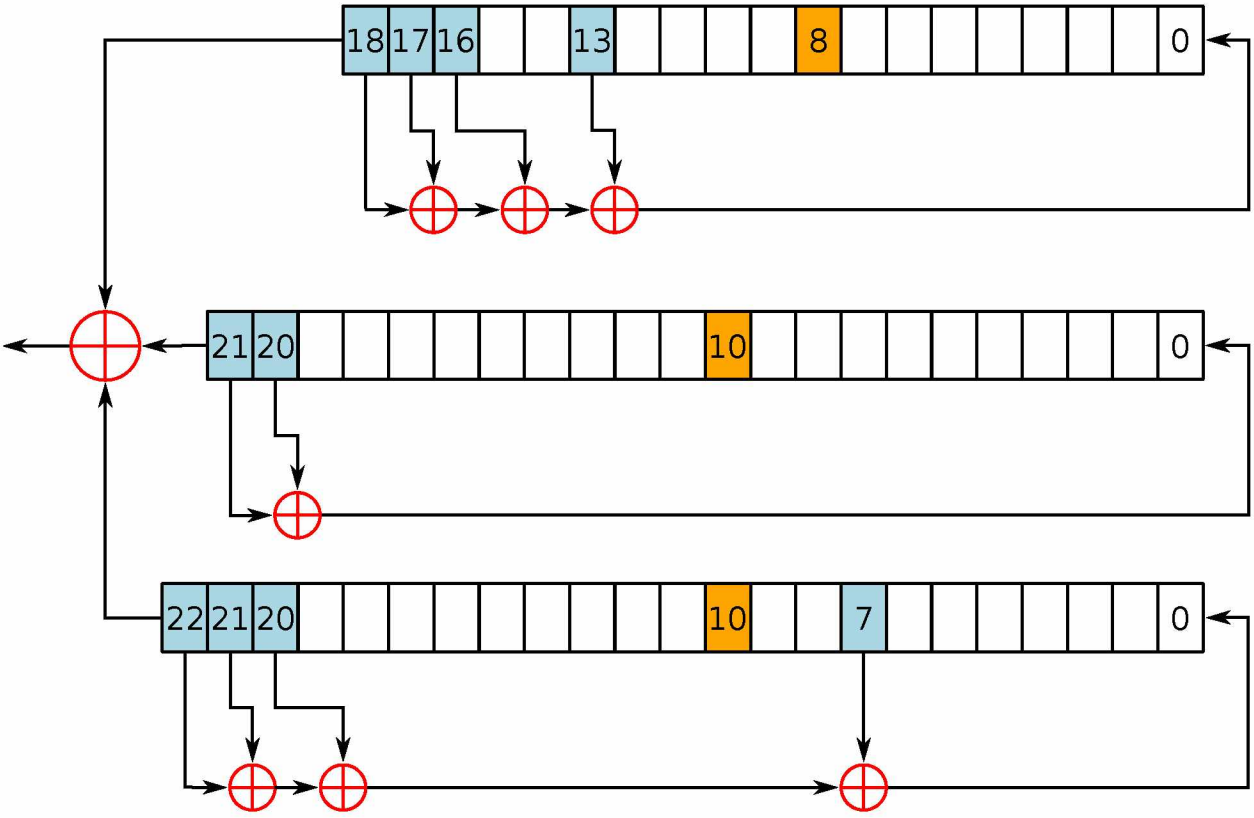


Figure 1: Layout of A5/1 cipher [12]

In Figure 1 the light blue squares represents the bits that are tapped in the respective LFSR. The orange square is the majority bit that is checked to determine the majority of all the shift registers. Lastly, the red + enclosed in a circle represents the XOR operation.

2.3 SIMD Within a Register

SWAR [5, 6] has other applications outside of cryptography. One use of it is to provide single instruction multiple data (SIMD) operations without requiring specialized hardware. For example adding four 8 bit numbers without SWAR can accomplished with four discrete additions. A serial C-style code example would look as follows.

```
a += a1; b += b1; c += c1; d += d1;
```

The diagram showing a visual layout is shown in Figure 2. The lines between the additions are used to show the separation between the operations.

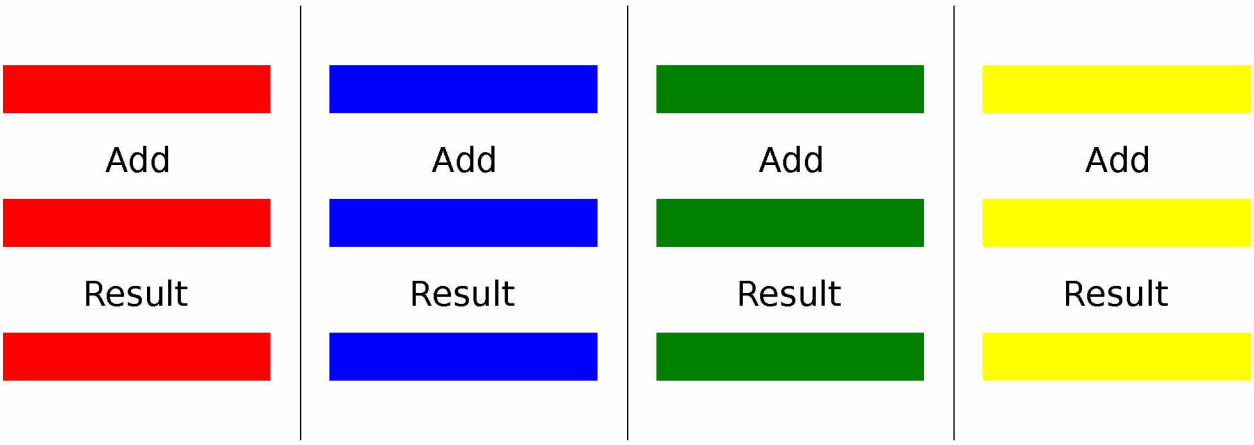


Figure 2: Serial 8-bit additions

Most architectures offer SIMD extensions to accomplish this operation with one SIMD instruction. However, this can also be accomplished without SIMD hardware by using SWAR. Another C-style code example for SWAR looks as follows [5, 6].

```
t = ((x & 0x7f7f7f7f) + (y & 0x7f7f7f7f));
t = (t ^ ((x ^ y) & 0x80808080));
```

In this example the four 8 bit values are packed into a single 32 bit machine register and it operates on all four values at once. However, it also turns the four operations from the serial example into six SWAR operations. Figure 3 shows how the SWAR four 8-bit additions would look.

Figure 3: SWAR 8-bit addition

If constraints can be placed on the packed values then a more efficient SWAR operation can occur. For example if only 7 bits are needed then they can be packed into the same 32 bit machine register with the spare bits used as padding between. This allows for the same four additions to occur in two operations [6]. This can be seen in Figure 4.

```
t = ((x + y) & 0x7f7f7f7f);
```

Again this effectively turns a 32 bit wide architecture into four 7-bit padded processors operating in parallel.

Figure 4: SWAR 7-bit addition

Figures 2, 3 and 4 show the difference in the layout between a serial implementation and a SWAR implementation of the addition of 4 numbers. The serial version operates on one 8-bit value at a time whereas the SWAR version operates on four of them in parallel. The same principle is applied to the LFSRs of the A5/1. Figure 5 shows the difference between the layout of serial LFSRs and the SWAR implementation. Specifically the colors dictate the grouping of the LFSR. Both implementations are operated on horizontally. The serial version only accesses a complete LFSR, while the SWAR version only accesses 1 bit of the LFSRs at a time.

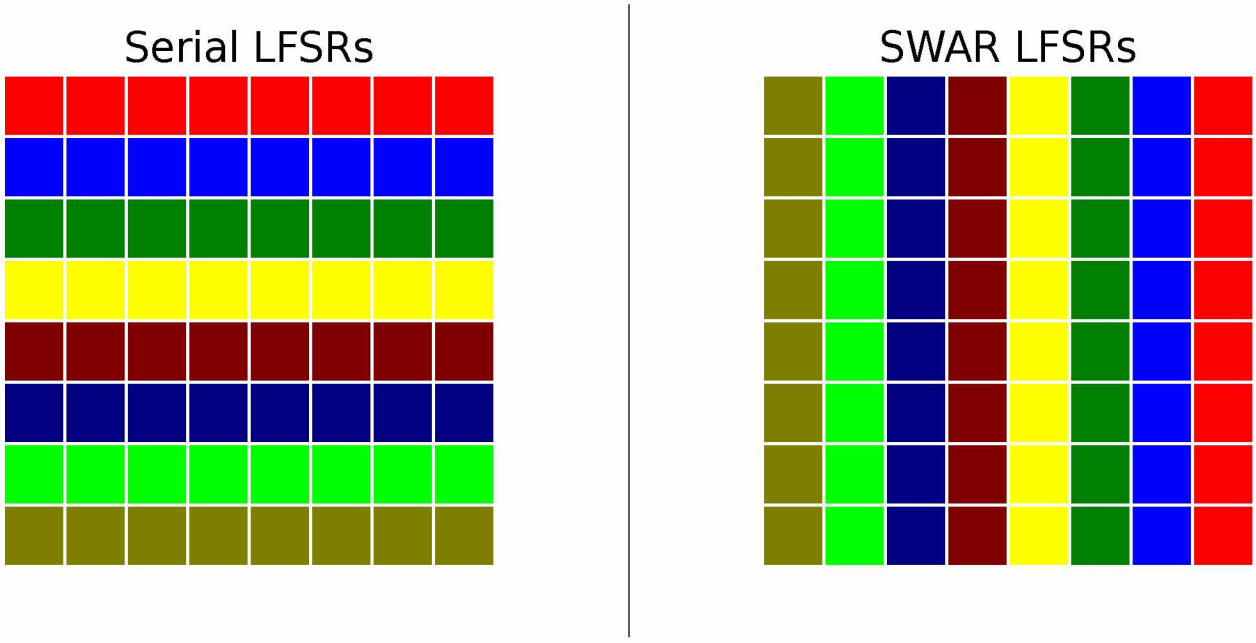


Figure 5: SWAR vs serial LFSRs

Moreover, the SWAR method makes use of general purpose registers. This means there are no special instructions or compiler intrinsics required to do these operations. The benefit of this is it is more or less portable to any platform/architecture as long as they support the width of the registers that are trying to be used [5]. An added benefit of this is that the program will not need to be rewritten to port to a different architecture, just recompiled.

2.4 Using SWAR on A5/1

The serial version of the A5/1 LFSRs can be held in 32 bit length machine registers (eg a C uint32_t or a register eax). They can be kept to the right length by using bit wise operations and masks. The clocking and feedback can also be simulated with bitwise operations as seen in the code below.

```
bit = ((lfsr >> 18) ^ (lfsr >> 17) ^ (lfsr >> 16) ^ (lfsr >> 13)) & 0x1;
lfsr = ((lfsr<<1) | bit) & MASK;
```

The code above simulates the first LFSR seen in the Table 1. Here the `<<` and `>>` mean shift left and right respectively, `^` means bitwise XOR, `&` means bitwise AND, and `|` bitwise OR. Here `bit` stores the value to be fed back into the LFSR named `lfsr`. Then the LFSR is clocked and the feedback value is put back on. The `MASK` is applied to keep the register to the specified length. The other two registers are done in a similar manner. The `MASK` needs to be changed according to the length of the registers, as well as the numeric constants to reflect the different tapped positions.

The SWAR implementation of the simulation of the same LFSR would look as follows.

```
uint64_t lowbits = lfsrs[18] ^ lfsrs[17] ^ lfsrs[16] ^ lfsrs[13];
for(int i = 18; i > 0; i--)
    lfsrs[i] = lfsrs[i-1];
lfsrs[0] = lowbits;
```

The SWAR version simulates 64 LFSRs at once. Meaning `lowbits` stores the value of the feedback for 64 LFSRs. The loop handles the clock by shifting the the values up one place in the LFSR. Once the shift is complete the value of the feedback is placed back on to the clocked LFSR.

The SWAR approach is straightforward with basic operations such as addition and LFSRs simulation. However, using the approach is more difficult when implementing more complex operations such as the majority clock in the A5/1 cipher. The truth table for the majority operation can be seen in Table 2. As there are three LFSRs the majority is what matches at least two out of the three. This operation can be deduced from boolean algebra and a minimal form can be found from use of a Karnaugh map (K-map).

Majority Function			
Reg1	Reg2	Reg3	Majority
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Table 2: A5/1 majority function

	ab			
c	00	01	11	10
0	0	0	1	0
1	0	1	1	1

Table 3: K-map for A5/1 majority function

Table 3 shows the K-map for the majority function. Using the K-map the function can be expressed in minterm canonical form. The minterm canonical form is $(a \wedge b) \vee (a \wedge c) \vee (b \wedge c)$, where a is Reg1, b is Reg2, and c is Reg3 from the truth table in Table 2. From here this translate directly into code. To compute the majority of the LFSRs using SWAR is shown below:

```
a = Reg1[8];
b = Reg2[10];
c = Reg3[10];
m = ((a & b) | (a & c) | (b & c));
```

The code calculates the majority of the LFSRs and stores them into temporary values. Since this is done SWAR style the majority for many LFSRs is stored into one variable. To get the majority bit of the first 16 LFSRs the code `m & 0xffff` could be used.

The SWAR version of the shift code differs substantially. The LFSRs are stored in a wider form and set to the desired length. The shifting happens with array operations. The irregular clocking and feedback shift is handled with a SIMD style if-then-else operation. Again this could be decomposed into boolean logic and minimized to the current form via K-maps.

```
for(i = s; i > 0; i--) {
    x[i] = ((x[i] & (~m)) | (x[i-1] & m));
}
x[0] = (x[0] & (~m)) | (n & m);
```

In this snippet `x` is one of the LFSRs, `m` is the result of XNOR of the majority and the clocking bits of the respective register, and `n` is the value that is to be fed back if a register is clocked. Here the loop handles the shift by moving the previous element in the LFSRs to the new location. If that register was not part of the majority and therefore not clocked then the shift does not occur. The rest of the operations are straightforward and are implemented with the bitwise operators in C.

2.5 CUDA

Traditionally the GPU was designed to perform both 2D and 3D graphics in realtime. However, with the release of CUDA in 2007 from NVIDIA this added functionality to GPUs that support CUDA to do more general purpose computations. The computations can be either serial or parallel [8]. To get the most out a GPU it is more suited for parallel computations. Moreover, CUDA can be used to help accelerate non-graphical computations, such as cryptography by an order of magnitude or more. The main trade off using CUDA is higher latency for higher throughput. Meaning the main intent is it not run one thread quickly, but running many threads concurrently [9].

The traditional CUDA execution model is to have the host, the CPU, setup the computation then instruct the device, the GPU, to process the computation. An example of this process is to have the CPU copy the data from main memory to the GPU. Then the CPU instructs the GPU to operate on the data and in turn the computation executes in parallel on GPU. Once the computation is finished the CPU copies the results back from GPU memory to main memory. There is an overhead associated with copying the data to and from the GPU. The way to offset the overhead is to have the GPU operate over large sets of data [8, 9].

CUDA differs from the traditional CPU programming model. CUDA has concepts of threads, warps, blocks, and grids. A CUDA thread is a single execution of a kernel. A warp is a group of 32 threads that all take the same branches. A block is group of threads that are executed together. A grid is a group of blocks that must finish execution before the program can continue. Lastly a kernel is like a regular function, but is executed N times in parallel by N different CUDA threads. CUDA also has a different memory hierarchy than that of a CPU. The number N is specified at the time of the kernel launch and it can not be changed once it is launched [9]. CUDA threads have private local memory that is only accessible to the thread. Blocks have special memory that is

shared between all threads in the block and has the same lifetime as the block. Constant and texture memory are also available. These forms of memory are more specialized and they are both read-only. Constant memory is used for data that will not change over the course of a kernel execution. Texture memory is optimized for 2D spatial locality. Lastly, there is global memory. All threads can access global memory. This is also where data is copied to from the main memory. The global, constant, and texture memory spaces are persistent across kernel launches by the same application [8].

Perhaps the easiest way to apply the parallelism CUDA provides is to do things that are naturally parallel. For instance, brute force key space enumeration. Here each CUDA thread could independently test all possible values in its partition of the key space. This also works well for bulk processing. Each CUDA thread could be assigned to do some unit of work independent of the other threads and continue until all the data has been processed.

```
/* CPU square elements in array */
void cpu_square(float *a, size_t len)
{
    for(int i = 0; i < len; i++)
        a[i] = a[i] * a[i];
}

/* CUDA square elements in array */
__global__ void cuda_square(float *a, size_t len)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if(idx < len)
        a[idx] = a[idx] * a[idx];
}
```

The code illustrates the difference between CUDA and CPU implementations of squaring the elements in an array. The CPU version happens serially and sequentially starting at the first element and continuing along until it reaches the end. The CUDA version squares the elements in parallel by computing the index of the thread that is running and performs the square operation. Here threadIdx.x is the current thread index that is running, blockIdx.x is the block in which the thread is located it, and blockDim.x is the number of threads per block. In the CUDA version the array is located in global memory.

2.6 Using CUDA on A5/1

To benefit from the massive parallelism available on GPUs the implementations must be ported to CUDA. The CUDA versions are very similar to the CPU versions of program. It has been modified so that it will run on a CUDA capable card. The main adjustment made was to have one CUDA thread simulate the LFSRs of the A5/1 cipher. This adjustment was made to both the serial and SWAR CPU versions. The main differences are the thread indexing that needs to be done so the CUDA threads know which data they need to operate on and the setup to get the data to the card. The interesting part of the SWAR version to CUDA is the indexing into memory based on which thread is running. The code for this is shown below:

```
lx = threadIdx.x + (blockIdx.x*blockDim.x);
kx = (threadIdx.x + (blockIdx.x*blockDim.x))*64;
fx = (threadIdx.x + (blockIdx.x*blockDim.x))*22;
```

Again threadIdx.x is the current thread index that is running, blockIdx.x is the current block that threadIdx.x is in, and blockDim.x is the number of threads per block. As it is setup each CUDA thread gets its own set of SWAR LFSRs. Each SWAR LFSRs needs its own key and frame. This is accomplished with an array that also stores the keys and frames in a vertical SWAR format as well. Each thread has a key that is 64 in length so the index into the array of keys needs to be adjusted accordingly. The same naturally follows for the frame.

3 Results

The program started as a straightforward serial A5/1 implementation. The serial A5/1 version simply simulates one set of A5/1 LFSRs at a time. This version can be seen in appendix A.1. From here it was changed to operate on the LFSRs using the SWAR method. In the SWAR A5/1 it simulates 64 different A5/1 LFSRs at once. As a result the SWAR versions are doing 64 times the amount of work that the serial versions are doing. The SWAR A5/1 implementation is seen in appendix A.2. Both of these versions are trivial to add CPU multithreading support by using OpenMP. This allows the A5/1 LFSRs to run in parallel on the CPU. For instance in the serial multithreaded implementation on a 4 core CPU up to 4 A5/1 LFSRs might be simulated at once. With the SWAR multithreaded version on a 4 core CPU there might be up to 256 A5/1 LFSRs being simulated. The CPU serial and SWAR multithreaded versions can be seen in appendix A.3 and A.4 respectively.

The CUDA versions are based off the CPU implementations. The serial CUDA version is like that of the CPU one. Each CUDA thread is assigned to simulate 1 set of LFSRs. As a result the number of sets of LFSRs being simulated at any given time is equal to the number of CUDA threads in execution. For example, if it was launched with 2048 CUDA threads then there could be up to 2048 sets of LFSRs being simulated. The implementation of the CUDA serial A5/1 is located in appendix A.5. The CPU SWAR version was also ported to run on a CUDA capable card. It has the same model as the serial CUDA version, that is each CUDA threads executes 1 SWAR set of LFSRs. Using the previous example of 2048 CUDA threads, this changes to the number of LFSRs being simulated at once to 131027. The CUDA SWAR A5/1 implementation is shown in appendix A.6.

As the code was only measured across varying Linux based operating systems the timing code is specific to that platform. The timing code header and body is shown in appendix A.7 and A.8 respectively. All execution times reported are from an average calculated from 1024 runs.

3.1 Equipment Used

The host computer specifications are as follows. Specifically the benchmarks were done with CUDA version 6.5 and compute architecture 3.0 hardware. Table 2 shows the specifications of the equipment used. Full specifications of the CPUs and GPUs can be found in Table 27.

	Computer 1	Computer 2	Computer 3
CPU	Intel Xeon E3 1275	Intel Celeron 2955U	Intel Xeon E3 1240 V2
GPU	GTX 670	N/A	N/A
RAM	16GB	4GB	32GB
OS	Ubuntu 14.04	Ubuntu 14.10	Ubuntu 14.04
CUDA	6.5	N/A	N/A
gcc	4.6.4	4.9.1	4.8.2

Table 4: Hardware used

The complete tables for execution times of all implementations run across all computers are located in Tables 28 to 41. Note, Computer 2 and Computer 3 do not have CUDA capable cards and therefore there is no timing data for the CUDA versions on these machines.

3.2 CPU Results

The timings results of the different implementations run on Computer 1 are visualized in Figure 6. The values which the graph was made from are reported in Tables 28 to 31. The legend in Figure 6 is further broken down as: A5/1 is the serial implementation, SA5/1 is the SWAR version, PA5/1 is the multithreaded serial version, and PSA5/1 is the multithreaded SWAR implementation.

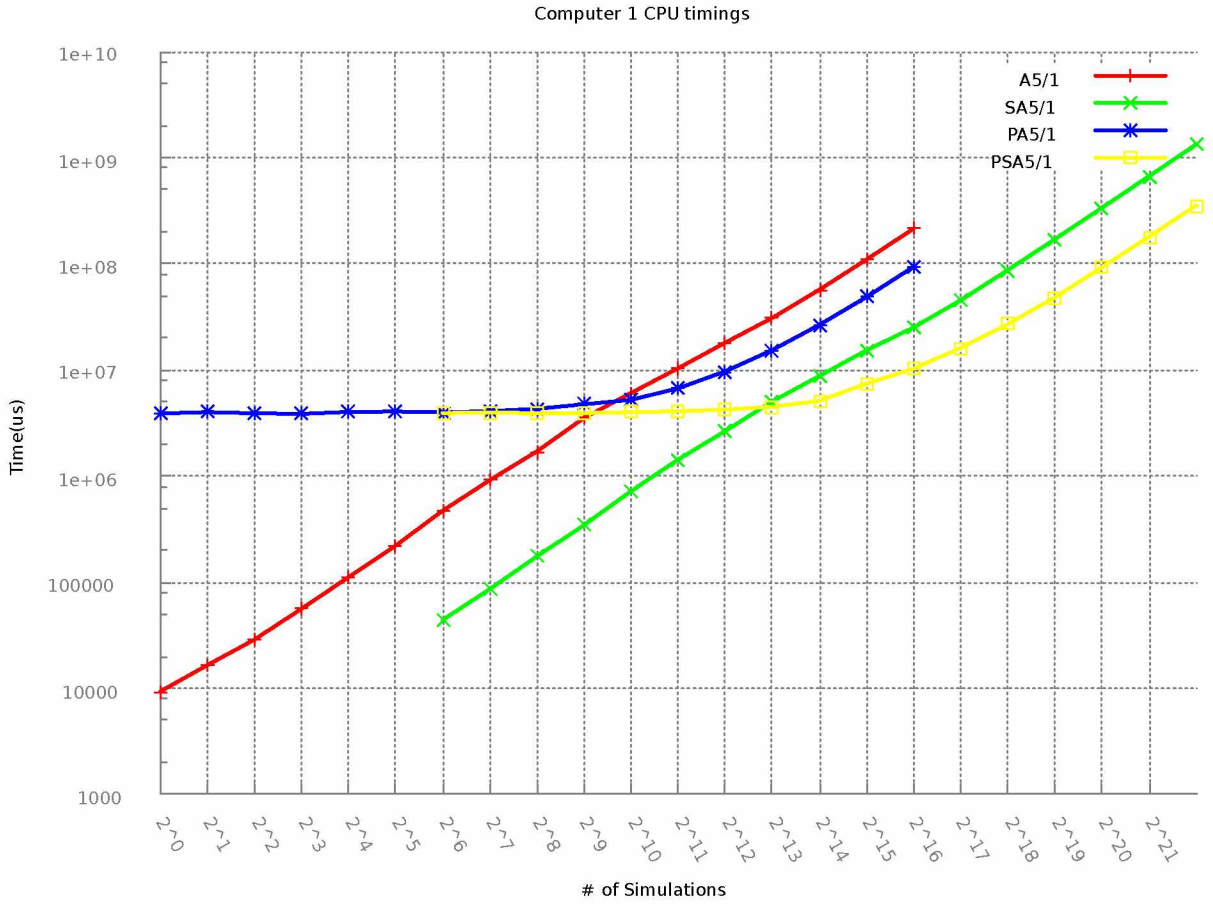


Figure 6: CPU implementations on Computer 1

Figure 6 also clearly shows the overhead of multithreading. The light blue and dark blue lines are dominated by the time it takes to create threads until the operating data set gets sufficiently large. From the graph sufficiently large appears to be 256 and 16384 simulations for the SWAR implementation, and serial implementation respectively.

The graphs for Computer 2 and Computer 3 look similar to that of Computer 1. This is not very surprising as they all share a common architecture and manufacturer. The timing graphs for Computer 2 and 3 are displayed in Figures 7 and 8.

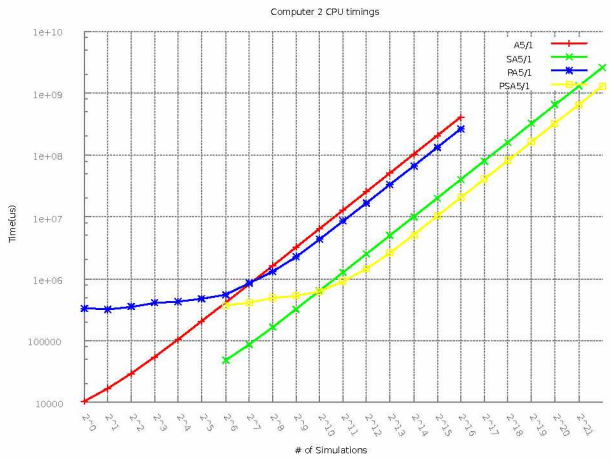


Figure 7: CPU results on Computer 2

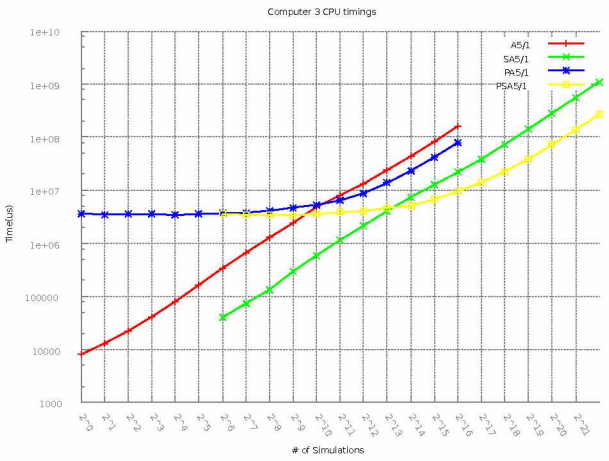


Figure 8: CPU results on Computer 3

The graphs of Computers 2 and 3 have very similar characteristics to that of Computer 1. In each computer the multithreaded versions are dominated by the overhead of thread creation until the data set gets large enough. Table 5 shows the factor of speed increased or decreased compared to the other implementations.

Computer 1 @ 1 Simulation			
A5/1 vs SA5/1	10.72	SA5/1 vs PA5/1	90.24
A5/1 vs PA5/1	0.00	SA5/1 vs PSA5/1	0.01
A5/1 vs PSA5/1	0.12	PA5/1 vs PSA5/1	1.02

Table 5: Factor speedup/slowdown at 1 simulation on Computer 1

From the Table 5 it can be seen that at 1 simulation the SWAR version outperformed the other implementations by at least a factor of 10. It managed a 90 fold speedup when compared to the non SWAR multithreaded version. Again, the main cause for this is the multithreaded versions have the overhead of creating threads which can not be effectively used on a data set this small.

Computer 1 @ 256 Simulations			
A5/1 vs SA5/1	9.75	SA5/1 vs PA5/1	24.58
A5/1 vs PA5/1	0.40	SA5/1 vs PSA5/1	1.17
A5/1 vs PSA5/1	0.44	PA5/1 vs PSA5/1	1.11

Table 6: Factor speedup/slowdown at 256 simulations on Computer 1

Table 6 displays the factor improvement or detriment across the differing implementations on Computer 1 doing 256 A5/1 simulations. Again, the non multithreaded SWAR implementation outperforms all other implementations.

Computer 1 @ 65535 Simulations			
A5/1 vs SA5/1	8.53	SA5/1 vs PA5/1	3.37
A5/1 vs PA5/1	2.29	SA5/1 vs PSA5/1	3.76
A5/1 vs PSA5/1	20.80	PA5/1 vs PSA5/1	9.08

Table 7: Factor speedup/slowdown at 65535 simulations on Computer 1

Once the data set get large enough the overhead of creating threads finally falls away. This can be seen in Table 7 which displays the the speed factor at 65536 simulations. It is shown that a multithreaded non SWAR implementation offers about a 2.3 speedup over a non multithreaded version. However, the multithreaded SWAR version offers over a 20 fold speedup compared to a serial non SWAR version.

So far the SWAR versions have provided significant speed improvements compared to the non SWAR implementations. Ideally as SWAR does 64 times the work it should also provide a 64 fold speedup. However, it is not possible to achieve the ideal speedup due to computer architecture, specifically the memory hierarchy. A more realistic ideal figure can be calculated though. The formula is as follows:

$$ideal = \frac{serial(N) * 64}{SWAR(N)}.$$

Here N is the timing results of the number of simulations that was run. The actual factor speedup is calculated as follows:

$$actual = \frac{serial(N * 64)}{SWAR(N)}.$$

Here the N remains the timing results of the number of simulations run, but the serial version uses the time that was measured for doing 64 times the work. Table 8 shows the calculated ideal factors for Computer 1 when doing 1 simulation.

1 Ideal Speedup Factor		
	SA5/1	PSA5/1
A5/1	13.56	0.15
PA5/1	5631.09	63.49

Table 8: Computer 1 Ideal Speedup Factor

1 Actual Speedup Factor		
	SA5/1	PSA5/1
A5/1	10.72	0.12
PA5/1	90.24	1.02

Table 9: Computer 1 Actual Speedup Factor

The factors in Table 8 show that the SWAR version has an ideal a factor of 13.56 speed increase. The actual increase is seen in 5, but is also shown in 9 for convenience. It can be seen we are just shy of the adjusted ideal speedup by only achieving a 10.72 fold. The multithreaded versions still compare very poorly at this small scale.

256 Ideal Speedup Factor		
	SA5/1	PSA5/1
A5/1	12.48	21.38
PA5/1	31.45	53.88

Table 10: Computer 1 Ideal Speed Factor

256 Actual Speedup Factor		
	SA5/1	PSA5/1
A5/1	9.75	0.44
PA5/1	24.58	1.11

Table 11: Computer 1 Actual Speed Factor

Table 10 has the adjusted ideal factors for 256 simulations. Again, the SWAR version expects about a 13 fold increase over the regular A5/1. The multithreaded SWAR version has an amazing ideal 21 and 53 fold increase over the non SWAR variants. However, looking at Table 11 it can be seen that the actual increase is either negative or much lower than ideal for the multithreaded versions as the thread overhead still dominates.

65536 Ideal Speedup Factor		
	SA5/1	PSA5/1
A5/1	10.29	38.68
PA5/1	4.49	16.89

Table 12: Computer 1 Ideal Speed Factor

65536 Actual Speedup Factor		
	SA5/1	PSA5/1
A5/1	8.53	20.80
PA5/1	3.73	9.08

Table 13: Computer 1 Actual Speed Factor

Finally, the adjusted ideal factors for 65536 simulations are shown in Table 12. Yet again the SWAR version is ideally going to have roughly 10 fold more performance than the non SWAR version. The actual adjusted factor in Table 13 shows that it only managed an 8.5 fold increase. The multithreaded versions are also expected to have a dramatic increase. The increase in the multithreaded SWAR version is ideally about 40 fold better than the serial version. As the data set is finally large enough to offset the overhead of the threads, it did manage to achieve about a 21 fold increase over the serial non SWAR version. It also managed a factor of 9 improvement over the multithreaded non SWAR version. This can mainly be attributed to the speedup that SWAR brings.

Computer 2 65536 Simulations Factors			
A5/1 vs SA5/1	10.20	SA5/1 vs PA5/1	6.59
A5/1 vs PA5/1	1.55	SA5/1 vs PSA5/1	9.68
A5/1 vs PSA5/1	19.94	PA5/1 vs PSA5/1	12.88

Table 14: Calculated Factors for Computer 2 at 65536 Simulations

Computer 2's timing factors for 65536 simulations is displayed in Table 14. Here it can be seen that the multithreaded non SWAR A5/1 performed about 1.5 fold better than the serial implementation. Again the SWAR variants perform much better than their non SWAR counterparts. The adjusted ideal increase is seen in Table 15 and, for convenience, the actual increase shown in Table 16. Again, the SWAR version offers about a 10 fold increase of that over the non SWAR.

65536 Ideal Speedup Factor		
	SA5/1	PSA5/1
A5/1	10.24	20.35
PA5/1	6.61	13.15

Table 15: Computer 2 Ideal Speed Factor

65536 Actual Speed Factor		
	SA5/1	PSA5/1
A5/1	10.20	19.94
PA5/1	6.59	12.88

Table 16: Computer 2 Actual Speed Factor

Lastly, Computer 3's timing factors are shown in Table 17. Again these are the factors for 65536 simulations. On this machine the non SWAR multithreaded version has a 2 fold increase over the non SWAR serial version. The SWAR version also compare poorly before adjustment with the serial and multithreaded versions giving about a 7 and 1.7 fold slowdown respectively.

Computer 3 65536 Simulations Factors			
A5/1 vs SA5/1	7.39	SA5/1 vs PA5/1	3.64
A5/1 vs PA5/1	2.03	SA5/1 vs PSA5/1	14.00
A5/1 vs PSA5/1	16.91	PA5/1 vs PSA5/1	8.32

Table 17: Calculated Factors for Computer 3 at 65536 Simulations

Taking the SWAR adjustments in to account it can be seen that the ideal speed up over the non SWAR version is about 9.3, which can be seen in Table 18. The multithreaded SWAR boasts an impressive ideal 38 fold increase over the non SWAR serial implementation. The actual fold increase, shown in Table 19, falls short of the ideal increase by only managing a 7.4 and 17 fold increase over the serial non SWAR. Note that the non multithreaded SWAR implementation outperforms the multithreaded non SWAR version by a factor of 3.6.

65536 Ideal Speedup Factor		
	SA5/1	PSA5/1
A5/1	9.29	38.13
PA5/1	4.57	18.76

Table 18: Computer 3 Ideal Speed Factor

65536 Actual Speedup Factor		
	SA5/1	PSA5/1
A5/1	7.39	16.91
PA5/1	3.64	8.32

Table 19: Computer 3 Actual Speed Factor

65536 Ideal Average Across Computers		
	SA5/1	PSA5/1
A5/1	9.94	32.39
PA5/1	5.22	12.79

Table 20: Average Ideal Speed Factor

65536 Actual Average Across Computers		
	SA5/1	PSA5/1
A5/1	8.71	19.22
PA5/1	4.65	10.09

Table 21: Average Actual Speed Factor

Tables 20 and 21 show the average fold increases when using SWAR over a non SWAR implementation. From the tables it is shown that on average SWAR offers an ideal increase of about 10 fold over non the non SWAR version. The actual measured increase is about a 9 fold increase. If multithreading can be utilized then a multithreaded SWAR has an ideal fold of about 32 over a serial non SWAR version and delivers an actual 19 fold improvement. Even a serial SWAR implementation manages a 4.6 fold increase over a multithreaded non SWAR version. It is not too surprising that the multithreaded SWAR implementation is about 10 fold more than the multithreaded non SWAR implementation, as SWAR seems to on average provide a 10 fold improvement.

3.3 CUDA Results

The timings results of the different CUDA implementations run on Computer 1 are visualized in Figure 9. Like CPU threads CUDA has an associated overhead that will dominate the timing results until the data set is large enough. This is from moving the data that will be operated on to and from the GPU. In this implementation the CUDA SWAR version needs 64 times the data that the CUDA non SWAR will need and as a result will have higher times due to more data being moved to and from the GPU.

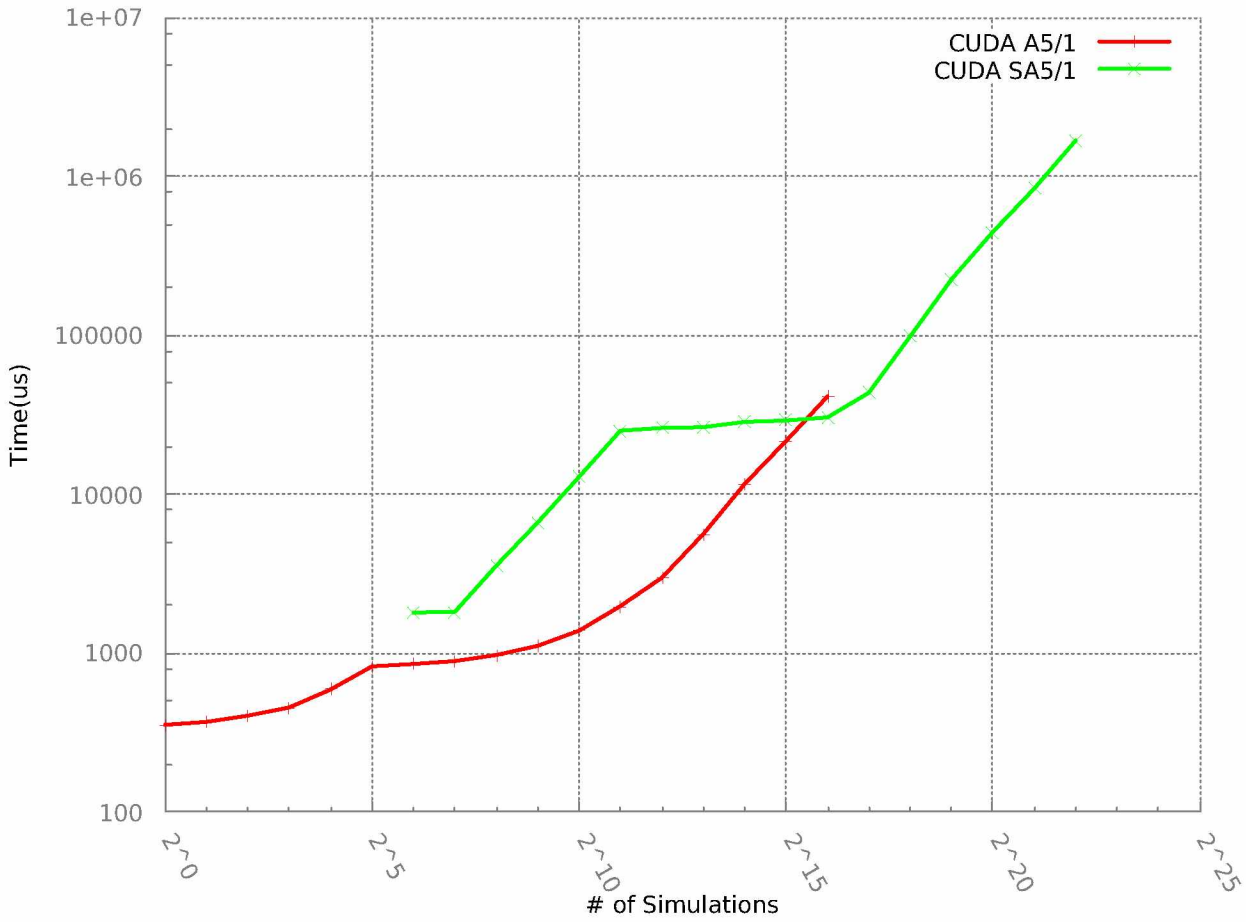


Figure 9: GPU implementations on Computer 1

Ideal CUDA Factor	
	CUDA SA5/1
CUDA A5/1	1.56

Table 22: Ideal CUDA Speed Factor

CUDA 65536 Simulations Factors	
	CUDA SA5/1
CUDA A5/1	1.35

Table 23: Actual CUDA Speed Factor

Table 23 shows the timing factors when comparing the CUDA A5/1 and the CUDA SWAR A5/1 implementations. Table 22 shows the ideal speedup using the same method as before. The actual speedup can be seen in Table 23. Here the SWAR method only managed a 1.35 fold speedup over the non SWAR version, making it just shy of the ideal speedup.

3.4 CPU vs GPU

The comparison in this context might be not completely fair due to varying levels of optimization between the implementations. However, both versions are at least modestly optimized so a compassion should not be completely skewed. Figure 10 shows the graphs of the execution times of the varying implementations, both CPU and GPU. The CPU results used here are from Computer 1.

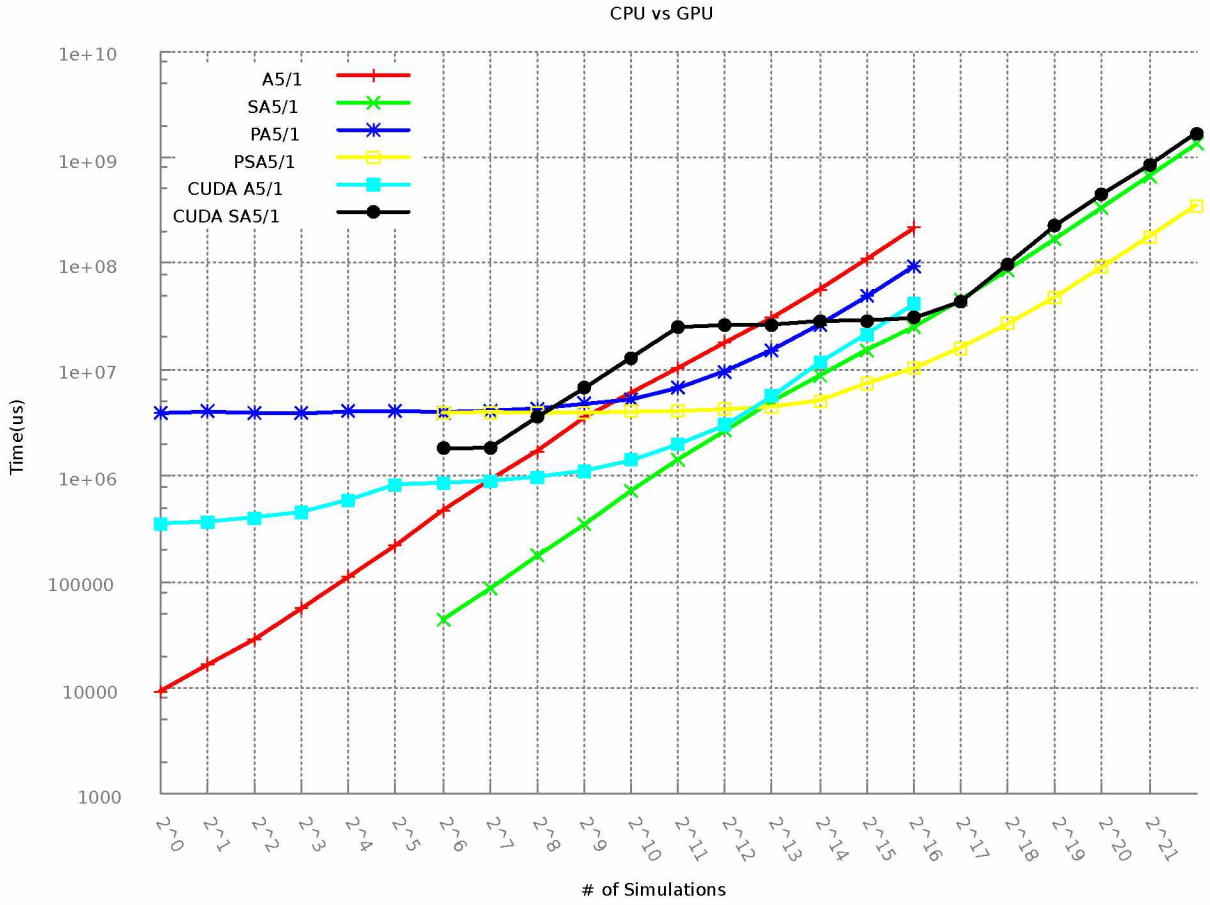


Figure 10: CPU vs GPU implementations

The graph in Figure 10 shows that once the data set gets large enough the multithreaded SWAR implementation has the smallest execution time. Table 24 shows the speed factors of the varying implementations compared. It is a left to right comparison, so whatever is on the right of the "vs" performed that much better than what is on the left.

CPU vs GPU speedup factors			
A5/1 vs CUDA A5/1	5.20	A5/1 vs CUDA SA5/1	7.02
PA5/1 vs CUDA A5/1	2.27	PA5/1 vs CUDA SA5/1	3.07
CUDA A5/1 vs SA5/1	1.64	CUDA SA5/1 vs SA5/1	1.27
CUDA A5/1 vs PSA5/1	4.00	CUDA SA5/1 vs PSA5/1	4.46
CUDA SA5/1 vs CUDA A5/1		1.35	

Table 24: CPU vs GPU at 65536 simulations

Ideal CPU vs GPU Factor	
	CUDA SA5/1
A5/1	8.12
PA5/1	3.55
CUDA A5/1	1.98

Table 25: Ideal CPU vs GPU Speed Factor

Actual CPU vs GPU Factor	
	CUDA SA5/1
A5/1	7.02
PA5/1	3.07
CUDA A5/1	1.35

Table 26: Actual CPU vs GPU Speed Factor

The adjusted ideal factor for the CUDA SWAR implementations are displayed in Table 25. It is expected that the CUDA SWAR implementation has an 8 fold better execution time over the CPU serial A5/1. Table 26 shows that the actual factor between the two is only 7.02. It is rather disappointing to see that the CUDA SWAR A5/1 only manages a 1.35 fold performance increase of the non SWAR CUDA A5/1 despite the fact that it is doing 64 times the work. It is interesting to see that once the data set gets large enough all the SWAR implementations start to outperform their non SWAR counterparts.

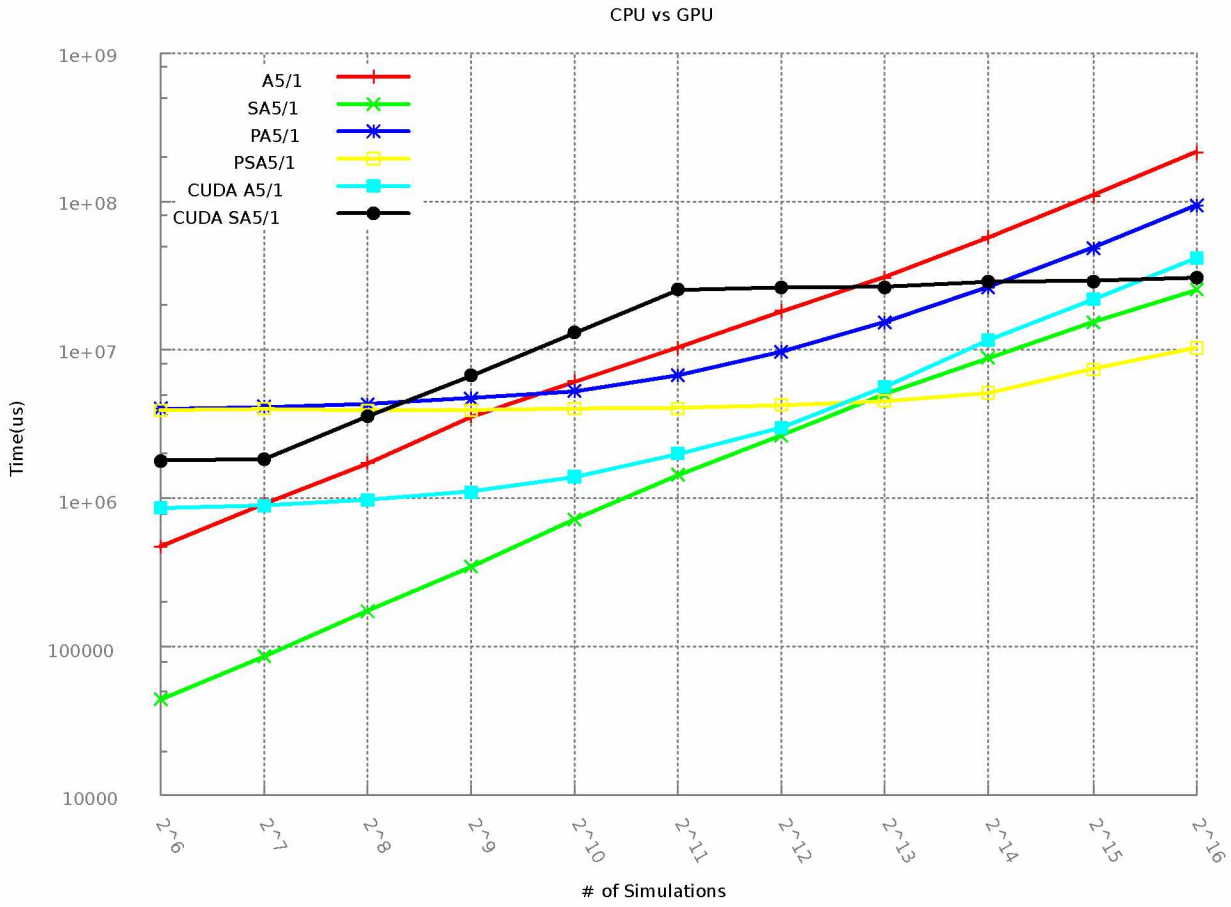


Figure 11: Zoomed and Adjusted CPU vs GPU implementations

It can be seen more clearly in Figure 11 the SWAR implementations out performing the non

SWAR counterparts. While the CUDA SWAR A5/1 outperforms non SWAR version it should be noted that both CPU SWAR implementations perform better than the CUDA versions. Table 24 shows the factors of the implementations compared. Here it can be seen that the CPU SWAR A5/1 managed a factor of about 1.3 over the CUDA SWAR A5/1. Moreover, the multithreaded CPU SWAR version managed about a 4.8 fold increase over the CUDA implementation.

4 Related Work

The technique of SWAR is not new invention and has various uses outside of cryptographic applications [10, 11]. When applied to cryptography SWAR is also referred to as bit-slicing. The first application of bit-slicing was used in a DES implementation for significant speedups. DES is a symmetric-key block cipher that uses 56-bit keys and 64-bit blocks. Using a similar SWAR technique a 3 to 5 fold speedup was achieved over standard DES on an Alpha processor [1].

As for related work involving the A5/1 cipher it revolves around cryptanalysis or attacks to break it. The attacks involve coming up with better than brute force solutions. The cipher was not a public standard and the implementation was reverse engineered. After which it has been subject to extensive cryptanalysis and number of serious weaknesses have been discovered [2, 7, 3].

5 Conclusion

The speedup that the simulation of linear feedback shift registers can gain from using SIMD within a register is high. As a result anything that uses LFSRs has the potential to benefit from such a gain. For example the A5/1 cipher used for voice privacy in GSM networks. Such gains can be useful for batch processing of a large data set, in brute force key enumeration, or handling multiple encrypted connections in parallel. This can be seen in the expected average 40 fold improvement that a multithreaded SWAR implementation has over the traditional serial non SWAR approach. The average 20 fold improvement that was actually achieved is still an impressive speed up. These improvements should be considered by anyone implementing something based on LFSRs.

6 References

[1] E. Biham. A fast new DES implementation in software. In *Fast Software Encryption*, pages 260–272. Springer, 1997.

[2] E. Biham and O. Dunkelman. Cryptanalysis of the A5/1 GSM stream cipher. In *Progress in Cryptology—INDOCRYPT 2000*, pages 43–51. Springer, 2000.

[3] A. Biryukov, A. Shamir, and D. Wagner. Real Time Cryptanalysis of A5/1 on a PC. In *Fast Software Encryption*, pages 1–18. Springer, 2001.

[4] M. Briceno, I. Goldberg, and D. Wagner. A pedagogical implementation of the GSM A5/1 and A5/2 "voice privacy" encryption algorithms. *Originally published at http://www. scard. org, mirror at http://cryptome. org/gsm-a512. htm*, 1999.

[5] R. J. Fisher. General-purpose SIMD within a register: Parallel processing on consumer microprocessors. 2003.

[6] R. J. Fisher and H. G. Dietz. Compiling for SIMD within a register. In *Languages and Compilers for Parallel Computing*, pages 290–305. Springer, 1999.

[7] K. Nohl. Attacking phone privacy. *Black Hat USA*, 2010.

[8] NVIDIA. CUDA Toolkit Documentation, 2014. [Online; accessed 21-November-2014].

[9] NVIDIA. Programming Guide :: CUDA Toolkit Documentation, 2014. [Online; accessed 21-November-2014].

[10] F. L. Padua, G. A. Pereira, P. JOSe, Q. Neto, M. F. Campos, and A. O. Fernades. Improving processing time of large images by instruction level parallelism. *CD-ROM Proc. Chilean Computing Week*, pages 5–9, 2001.

[11] L. A. Spracklen. *SWAR systems and communications applications.* PhD thesis, University of Aberdeen, 2001.

[12] Wikipedia. A5-1 GSM encryption, 2009. [Online; accessed 16-October-2014].

A Appendix

A.1 a5.c

```
/*
    Karl Ott
    CPU A5/1 cipher
*/

#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <time.h>
#include <stdlib.h>
#include "timing.h"

/*
    indexed as lsb as 0
    a = 19 bits in length tapped at bits 13,16,17,18 clocked on bit 8
    b = 22 bits in length tapped at bits 20,21 clocked on bit 10
    c = 23 bits in length tapped at bits 7,20,21,22 clocked on bit 10
    polynomials:
        x^19 + x^18 + x^17 + x^14 + 1
        x^22 + x^21 + 1
        x^23 + x^22 + x^21 + x^8 + 1
    initalized to 0
    64 bit key is xored into the lsb of the registers
    22 bit frame is then xored into the lsb of the registers
    100 majority clocks follow with discarded output
    registers are clocked using a majority rule
    registers are now ready to produce two 114 bit sequences, down/up
    as per wikipedia: http://en.wikipedia.org/wiki/A5/1
*/

enum {
    MASKA = 0x0007ffff,
    MASKB = 0x003fffff,
    MASKC = 0x007fffff,
    TAPA = 0x00072000,
    TAPB = 0x00300000,
    TAPC = 0x00700080,
    CLKA = 0x00000100,
    CLKB = 0x00000400,
    CLKC = 0x00000400,
    OUTA = 0x00040000,
    OUTB = 0x00200000,
    OUTC = 0x00400000
};

struct lfsrs {
    uint32_t a,b,c;
};
```

```
uint32_t
majority(struct lfsrs x) {
    uint32_t a = (x.a & CLKA) >> 8;
    a += (x.b & CLKB) >> 10;
    a += (x.c & CLKC) >> 10;
    return ((a & 0x2)>>1);
}

uint32_t
clockreg(uint32_t x, uint32_t mask, uint32_t tap) {
    uint32_t a = 0;
    switch(tap) {
    case TAPA:
        a = ((x >> 18) ^ (x >> 17) ^ (x >> 16) ^ (x >> 13)) & 0x1;
        break;
    case TAPB:
        a = ((x >> 21) ^ (x >> 20)) & 0x1;
        break;
    case TAPC:
        a = ((x >> 22) ^ (x >> 21) ^ (x >> 20) ^ (x >> 7)) & 0x1;
        break;
    default:
        a = 0;
        break;
    }
    return (mask & ((x << 1) | a));
}

void
clocklfsrs(struct lfsrs * x) {
    x->a = clockreg(x->a, MASKA, TAPA);
    x->b = clockreg(x->b, MASKB, TAPB);
    x->c = clockreg(x->c, MASKC, TAPC);
}

void
clockmaj(struct lfsrs * x) {
    uint32_t maj = majority(*x);
    if(maj == ((x->a & CLKA) != 0)) {
        x->a = clockreg(x->a, MASKA, TAPA);
    }
    if(maj == ((x->b & CLKB) != 0)) {
        x->b = clockreg(x->b, MASKB, TAPB);
    }
    if(maj == ((x->c & CLKC) != 0)) {
        x->c = clockreg(x->c, MASKC, TAPC);
    }
}

uint32_t
highbit(struct lfsrs x) {
    return ((x.a & OUTA) >> 18) ^ ((x.b & OUTB) >> 21) ^ ((x.c & OUTC) >> 22);
}
```

```
void
setup(struct lfsrs * x, uint64_t key, uint32_t frame) {
    uint32_t i, t;
    for(i = 0; i < 64; i++) {
        clocklfsrs(x);
        t = (key >> i) & 1;
        x->a ^= t;
        x->b ^= t;
        x->c ^= t;
    }
    for(i = 0; i < 22; i++) {
        clocklfsrs(x);
        t = (frame >> i) & 1;
        x->a ^= t;
        x->b ^= t;
        x->c ^= t;
    }
    for(i = 0; i < 100; i++) {
        clockmaj(x);
    }
}

void
run(struct lfsrs * x, uint8_t * a, uint8_t * b) {
    uint32_t i, h;
    for(i = 0; i < 114; i++) {
        clockmaj(x);
        h = highbit(*x);
        a[i/8] |= h << (7 - (i&7));
    }
    for(i = 0; i < 114; i++) {
        clockmaj(x);
        h = highbit(*x);
        b[i/8] |= h << (7 - (i&7));
    }
}

int
main(int argc, char *argv[]) {
    struct lfsrs * regs;
    struct timespec start, end, total;
    uint64_t * key;
    uint32_t * frame;
    uint32_t in, i;
    uint8_t a[15];  uint8_t b[15];

    if(argc < 2) {
        printf("incorrect␣usage:␣./a5␣numsims\n");
        exit(-1);
    }

    in = atoi(argv[1]);

    regs = malloc(sizeof(struct lfsrs) * in);
```

```
    memset(regs, 0, sizeof(struct lfsrs) * in);
    key = malloc(sizeof(uint64_t) * 64 * in);
    frame = malloc(sizeof(uint64_t) * 22 * in);
    key[0] = 0xefcdab8967452312;
    frame[0] = 0x134;
    memset(a, 0, sizeof(a));
    memset(b, 0, sizeof(b));

    clock_gettime(CLOCK_MONOTONIC, &start);
    for(i = 0; i < in; i++) {
        setup(&regs[i], key[i], frame[i]);
        run(&regs[i], a, b);
    }
    clock_gettime(CLOCK_MONOTONIC, &end);
    total = diff(start, end);

    printf("took:␣%ld␣to␣run\n", total.tv_sec*1000000000+total.tv_nsec);

    return 0;
}
```

A.2 sa5.c

```
/*
    Karl Ott
    CPU SWAR A5/1 cipher
*/

#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include "timing.h"

/*
    indexed as lsb as 0
    a = 19 bits in length tapped at bits 13,16,17,18 clocked on bit 8
    b = 22 bits in length tapped at bits 20,21 clocked on bit 10
    c = 23 bits in length tapped at bits 7,20,21,22 clocked on bit 10
    polynomials:
        x^19 + x^18 + x^17 + x^14 + 1
        x^22 + x^21 + 1
        x^23 + x^22 + x^21 + x^8 + 1
    initalized to 0
    64 bit key is xored into the lsb of the registers
    22 bit frame is then xored into the lsb of the registers
    100 majority clocks follow with discarded output
    registers are clocked using a majority rule
    registers are now ready to produce two 114 bit sequences, down/up
    as per wikipedia: http://en.wikipedia.org/wiki/A5/1
*/

#define ALL 0xffffffffffffffff

enum {
    TAPA = 0,
    TAPB = 1,
    TAPC = 2
};

struct lfsrs {
    uint64_t a[19];
    uint64_t b[22];
    uint64_t c[23];
};

/*
 * Below calculates the majority for each set of registers.
 * So each bit corresponds to the majority for the 3 registers.
 */
uint64_t
majority(struct lfsrs * x) {
    uint64_t i,j,k;
    i = x->a[8];
```

```
    j = x->b[10];
    k = x->c[10];
    return ((j & k) | (i & k) | (i & j));
}

void
clockreg(uint64_t * x, uint32_t t, uint64_t m) {
    uint64_t s, n, i;
    switch(t) {
    case TAPA:
        s = 18;
        n = (x[13] ^ x[16] ^ x[17] ^ x[18]);
        break;
    case TAPB:
        s = 21;
        n = (x[20] ^ x[21]);
        break;
    case TAPC:
        s = 22;
        n = (x[7] ^ x[20] ^ x[21] ^ x[22]);
        break;
    default:
        printf("%s","uhh, we shouldn't be here!\n");
        s = 0;
        m = 0;
        n = 0;
        break;
    }

    for(i = s; i > 0; i--) {
        x[i] = ((x[i] & (~m)) | (x[i-1] & m));
    }
    x[0] = (x[0] & (~m)) | (n & m);
}

void
clocklfsrs(struct lfsrs * x) {
    clockreg(x->a, TAPA, ALL);
    clockreg(x->b, TAPB, ALL);
    clockreg(x->c, TAPC, ALL);
}

void
clockmaj(struct lfsrs * x) {
    uint64_t maj = majority(x);
    clockreg(x->a, TAPA, ~(maj ^ x->a[8]));
    clockreg(x->b, TAPB, ~(maj ^ x->b[10]));
    clockreg(x->c, TAPC, ~(maj ^ x->c[10]));
}

uint64_t
highbits(struct lfsrs * x) {
    return (x->a[18] ^ x->b[21] ^ x->c[22]);
}
```

```
void
setup(struct lfsrs * x, uint64_t * key, uint64_t * frame) {
    uint32_t i;
    for(i = 0; i < 64; i++) {
        clocklfsrs(x);
        x->a[0] ^= key[i];
        x->b[0] ^= key[i];
        x->c[0] ^= key[i];
    }
    for(i = 0; i < 22; i++) {
        clocklfsrs(x);
        x->a[0] ^= frame[i];
        x->b[0] ^= frame[i];
        x->c[0] ^= frame[i];
    }
    for(i = 0; i < 100; i++) {
        clockmaj(x);
    }
}

void
run(struct lfsrs * x, uint64_t * a, uint64_t * b) {
    uint32_t i;
    for(i = 0; i < 114; i++) {
        clockmaj(x);
        a[i] = highbits(x);
    }
    for(i = 0; i < 114; i++) {
        clockmaj(x);
        b[i] = highbits(x);
    }
}

int
main(int argc, char *argv[]) {
    struct lfsrs * regs;
    struct timespec start, end, total;
    uint64_t k;
    uint64_t * key;
    uint64_t * frame;
    uint64_t f, i;
    uint32_t in;
    uint64_t a[114], b[114];

    if(argc < 2) {
        printf("incorrect␣usage:␣./sa5␣numsims\n");
        exit(-1);
    }

    in = atoi(argv[1]);

    regs = malloc(sizeof(struct lfsrs) * in);
    memset(regs, 0, sizeof(struct lfsrs) * in);
```

```
    key = malloc(sizeof(uint64_t) * 64 * in);
    frame = malloc(sizeof(uint64_t) * 22 * in);
    memset(a, 0, sizeof(a));
    memset(b, 0, sizeof(b));
    k = 0xefcdab8967452312;
    f = 0x134;
    for(i = 0; i < 64; i++) {
        key[i] = (k >> i) & 1;
    }

    for(i = 0; i < 22; i++) {
        frame[i] = (f >> i) & 1;
    }


    clock_gettime(CLOCK_MONOTONIC, &start);
    for(i = 0; i < in; i++) {
        setup(&regs[i], &key[i*64], &frame[i*22]);
        run(&regs[i], a, b);
    }
    clock_gettime(CLOCK_MONOTONIC, &end);
    total = diff(start, end);

    printf("took:␣%ld␣to␣run\n", total.tv_sec*1000000000+total.tv_nsec);

    return 0;
}
```

A.3 pa5.c

```
/*
    Karl Ott
    Multithreaded CPU A5/1 cipher
*/

#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <time.h>
#include <omp.h>
#include <stdlib.h>
#include "timing.h"

/*
    indexed as lsb as 0
    a = 19 bits in length tapped at bits 13,16,17,18 clocked on bit 8
    b = 22 bits in length tapped at bits 20,21 clocked on bit 10
    c = 23 bits in length tapped at bits 7,20,21,22 clocked on bit 10
    polynomials:
        x^19 + x^18 + x^17 + x^14 + 1
        x^22 + x^21 + 1
        x^23 + x^22 + x^21 + x^8 + 1
    initalized to 0
    64 bit key is xored into the lsb of the registers
    22 bit frame is then xored into the lsb of the registers
    100 majority clocks follow with discarded output
    registers are clocked using a majority rule
    registers are now ready to produce two 114 bit sequences, down/up
    as per wikipedia: http://en.wikipedia.org/wiki/A5/1
*/

enum {
    MASKA = 0x0007ffff,
    MASKB = 0x003fffff,
    MASKC = 0x007fffff,
    TAPA = 0x00072000,
    TAPB = 0x00300000,
    TAPC = 0x00700080,
    CLKA = 0x00000100,
    CLKB = 0x00000400,
    CLKC = 0x00000400,
    OUTA = 0x00040000,
    OUTB = 0x00200000,
    OUTC = 0x00400000
};

struct lfsrs {
    uint32_t a,b,c;
};

uint32_t
majority(struct lfsrs x) {
```

```
    uint32_t a = (x.a & CLKA) >> 8;
    a += (x.b & CLKB) >> 10;
    a += (x.c & CLKC) >> 10;
    return ((a & 0x2)>>1);
}

uint32_t
clockreg(uint32_t x, uint32_t mask, uint32_t tap) {
    uint32_t a = 0;
    switch(tap) {
    case TAPA:
        a = ((x >> 18) ^ (x >> 17) ^ (x >> 16) ^ (x >> 13)) & 0x1;
        break;
    case TAPB:
        a = ((x >> 21) ^ (x >> 20)) & 0x1;
        break;
    case TAPC:
        a = ((x >> 22) ^ (x >> 21) ^ (x >> 20) ^ (x >> 7)) & 0x1;
        break;
    }
    return(mask & ((x << 1) | a));
}

void
clocklfsrs(struct lfsrs * x) {
    x->a = clockreg(x->a, MASKA, TAPA);
    x->b = clockreg(x->b, MASKB, TAPB);
    x->c = clockreg(x->c, MASKC, TAPC);
}

void
clockmaj(struct lfsrs * x) {
    uint32_t maj = majority(*x);
    if(maj == ((x->a & CLKA) != 0)) {
        x->a = clockreg(x->a, MASKA, TAPA);
    }
    if(maj == ((x->b & CLKB) != 0)) {
        x->b = clockreg(x->b, MASKB, TAPB);
    }
    if(maj == ((x->c & CLKC) != 0)) {
        x->c = clockreg(x->c, MASKC, TAPC);
    }
}

uint32_t
highbit(struct lfsrs x) {
    return ((x.a & OUTA) >> 18) ^ ((x.b & OUTB) >> 21) ^ ((x.c & OUTC) >> 22);
}

void
setup(struct lfsrs * x, uint64_t key, uint32_t frame) {
    uint32_t i, t;
    for(i = 0; i < 64; i++) {
        clocklfsrs(x);
```

```
        t = (key >> i) & 1;
        x->a ^= t;
        x->b ^= t;
        x->c ^= t;
    }
    for(i = 0; i < 22; i++) {
        clocklfsrs(x);
        t = (frame >> i) & 1;
        x->a ^= t;
        x->b ^= t;
        x->c ^= t;
    }
    for(i = 0; i < 100; i++) {
        clockmaj(x);
    }
}

void
run(struct lfsrs * x, uint8_t * a, uint8_t * b) {
    uint32_t i, h;
    for(i = 0; i < 114; i++) {
        clockmaj(x);
        h = highbit(*x);
        a[i/8] |= h << (7 - (i&7));
    }
    for(i = 0; i < 114; i++) {
        clockmaj(x);
        h = highbit(*x);
        b[i/8] |= h << (7 - (i&7));
    }
}

int
main(int argc, char *argv[]) {
    struct lfsrs * regs;
    struct timespec start, end, total;
    uint64_t * key;
    uint32_t * frame;
    uint32_t in, i;
    uint8_t a[15];  uint8_t b[15];

    if(argc < 2) {
        printf("incorrect␣usage:␣./pa5␣numsims\n");
        exit(-1);
    }

    in = atoi(argv[1]);

    regs = malloc(sizeof(struct lfsrs) * in);
    memset(regs, 0, sizeof(struct lfsrs) * in);
    key = malloc(sizeof(uint64_t) * 64 * in);
    frame = malloc(sizeof(uint64_t) * 22 * in);
    key[0] = 0xefcdab8967452312;
    frame[0] = 0x134;
```

```
    memset(a, 0, sizeof(a));
    memset(b, 0, sizeof(b));

    clock_gettime(CLOCK_MONOTONIC, &start);
    #pragma omp parallel for
    for(i = 0; i < in; i++) {
        setup(&regs[i], key[i], frame[i]);
        run(&regs[i], a, b);
    }
    clock_gettime(CLOCK_MONOTONIC, &end);
    total = diff(start, end);

    printf("took:␣%ld␣to␣run\n", total.tv_sec*1000000000+total.tv_nsec);

    return 0;
}
```

A.4 psa5.c

```
/*
    Karl Ott
    Multithreaded CPU SWAR A5/1 cipher
*/

#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>
#include "timing.h"

/*
    indexed as lsb as 0
    a = 19 bits in length tapped at bits 13,16,17,18 clocked on bit 8
    b = 22 bits in length tapped at bits 20,21 clocked on bit 10
    c = 23 bits in length tapped at bits 7,20,21,22 clocked on bit 10
    polynomials:
        x^19 + x^18 + x^17 + x^14 + 1
        x^22 + x^21 + 1
        x^23 + x^22 + x^21 + x^8 + 1
    initalized to 0
    64 bit key is xored into the lsb of the registers
    22 bit frame is then xored into the lsb of the registers
    100 majority clocks follow with discarded output
    registers are clocked using a majority rule
    registers are now ready to produce two 114 bit sequences, down/up
    as per wikipedia: http://en.wikipedia.org/wiki/A5/1
*/

#define ALL 0xffffffffffffffff

enum {
    TAPA = 0,
    TAPB = 1,
    TAPC = 2
};

struct lfsrs {
    uint64_t a[19];
    uint64_t b[22];
    uint64_t c[23];
};

/*
 * Below calculates the majority for each set of registers.
 * So each bit corresponds to the majority for the 3 registers.
 */
uint64_t
majority(struct lfsrs * x) {
    uint64_t i,j,k;
```

```
    i = x->a[8];
    j = x->b[10];
    k = x->c[10];
    return ((j & k) | (i & k) | (i & j));
}

void
clockreg(uint64_t * x, uint32_t t, uint64_t m) {
    uint64_t s, n, i;
    switch(t) {
    case TAPA:
        s = 18;
        n = (x[13] ^ x[16] ^ x[17] ^ x[18]);
        break;
    case TAPB:
        s = 21;
        n = (x[20] ^ x[21]);
        break;
    case TAPC:
        s = 22;
        n = (x[7] ^ x[20] ^ x[21] ^ x[22]);
        break;
    default:
        printf("%s","uhh, we shouldn't be here!\n");
        s = 0;
        m = 0;
        n = 0;
        break;
    }

    for(i = s; i > 0; i--) {
        x[i] = ((x[i] & (~m)) | (x[i-1] & m));
    }
    x[0] = (x[0] & (~m)) | (n & m);
}

void
clocklfsrs(struct lfsrs * x) {
    clockreg(x->a, TAPA, ALL);
    clockreg(x->b, TAPB, ALL);
    clockreg(x->c, TAPC, ALL);
}

void
clockmaj(struct lfsrs * x) {
    uint64_t maj = majority(x);
    clockreg(x->a, TAPA, ~(maj ^ x->a[8]));
    clockreg(x->b, TAPB, ~(maj ^ x->b[10]));
    clockreg(x->c, TAPC, ~(maj ^ x->c[10]));
}

uint64_t
highbits(struct lfsrs * x) {
    return (x->a[18] ^ x->b[21] ^ x->c[22]);
```

```
}

void
setup(struct lfsrs * x, uint64_t * key, uint64_t * frame) {
    uint32_t i;
    for(i = 0; i < 64; i++) {
        clocklfsrs(x);
        x->a[0] ^= key[i];
        x->b[0] ^= key[i];
        x->c[0] ^= key[i];
    }
    for(i = 0; i < 22; i++) {
        clocklfsrs(x);
        x->a[0] ^= frame[i];
        x->b[0] ^= frame[i];
        x->c[0] ^= frame[i];
    }
    for(i = 0; i < 100; i++) {
        clockmaj(x);
    }
}

void
run(struct lfsrs * x, uint64_t * a, uint64_t * b) {
    uint32_t i;
    for(i = 0; i < 114; i++) {
        clockmaj(x);
        a[i] = highbits(x);
    }
    for(i = 0; i < 114; i++) {
        clockmaj(x);
        b[i] = highbits(x);
    }
}

int
main(int argc, char *argv[]) {
    struct lfsrs * regs;
    struct timespec start, end, total;
    uint64_t k;
    uint64_t * key;
    uint64_t * frame;
    uint64_t f, i;
    uint32_t in;
    uint64_t a[114], b[114];

    if(argc < 2) {
        printf("incorrect␣usage:␣./psa5␣numsims\n");
        exit(-1);
    }

    in = atoi(argv[1]);

    regs = malloc(sizeof(struct lfsrs) * in);
```

```
    memset(regs, 0, sizeof(struct lfsrs) * in);
    key = malloc(sizeof(uint64_t) * 64 * in);
    frame = malloc(sizeof(uint64_t) * 22 * in);
    memset(a, 0, sizeof(a));
    memset(b, 0, sizeof(b));
    k = 0xefcdab8967452312;
    f = 0x134;
    for(i = 0; i < 64; i++) {
        key[i] = (k >> i) & 1;
    }

    for(i = 0; i < 22; i++) {
        frame[i] = (f >> i) & 1;
    }


    clock_gettime(CLOCK_MONOTONIC, &start);
    #pragma omp parallel for
    for(i = 0; i < in; i++) {
        setup(&regs[i], &key[i*64], &frame[i*22]);
        run(&regs[i], a, b);
    }
    clock_gettime(CLOCK_MONOTONIC, &end);
    total = diff(start, end);

    printf("took:␣%ld␣to␣run\n", total.tv_sec*1000000000+total.tv_nsec);

    return 0;
}
```

A.5 cudaa5.cu

```
/*
    Karl Ott
    CUDA A5/1 cipher
*/

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <cuda.h>
#include <time.h>
#include "timing.h"

/*
    indexed as lsb as 0
    a = 19 bits in length tapped at bits 13,16,17,18 clocked on bit 8
    b = 22 bits in length tapped at bits 20,21 clocked on bit 10
    c = 23 bits in length tapped at bits 7,20,21,22 clocked on bit 10
    polynomials:
        x^19 + x^18 + x^17 + x^14 + 1
        x^22 + x^21 + 1
        x^23 + x^22 + x^21 + x^8 + 1
    initalized to 0
    64 bit key is xored into the lsb of the registers
    22 bit frame is then xored into the lsb of the registers
    100 majority clocks follow with discarded output
    registers are clocked using a majority rule
    registers are now ready to produce two 114 bit sequences, down/up
    as per wikipedia: http://en.wikipedia.org/wiki/A5/1
*/

enum {
    MASKA = 0x0007ffff,
    MASKB = 0x003fffff,
    MASKC = 0x007fffff,
    TAPA = 0x00072000,
    TAPB = 0x00300000,
    TAPC = 0x00700080,
    CLKA = 0x00000100,
    CLKB = 0x00000400,
    CLKC = 0x00000400,
    OUTA = 0x00040000,
    OUTB = 0x00200000,
    OUTC = 0x00400000
};


struct lfsrs {
    uint32_t a;
    uint32_t b;
    uint32_t c;
};
```

```
/*
 * Below calculates the majority for each set of registers.
 * So each bit corresponds to the majority for the 3 registers.
 */
__device__ uint32_t
majority(struct lfsrs * x) {
    uint32_t a = (x->a & CLKA) >> 8;
    a += (x->b & CLKB) >> 10;
    a += (x->c & CLKC) >> 10;
    return ((a & 0x2)>>1);
}

__device__ uint32_t
clockreg(uint32_t x, uint32_t mask, uint32_t tap) {
    uint32_t a = 0;
    switch(tap) {
    case TAPA:
        a = ((x >> 18) ^ (x >> 17) ^ (x >> 16) ^ (x >> 13)) & 0x1;
        break;
    case TAPB:
        a = ((x >> 21) ^ (x >> 20)) & 0x1;
        break;
    case TAPC:
        a = ((x >> 22) ^ (x >> 21) ^ (x >> 20) ^ (x >> 7)) & 0x1;
        break;
    default:
        a = 0;
        break;
    }
    return (mask & ((x << 1) | a));
}

__device__ void
clocklfsrs(struct lfsrs * x) {
    x->a = clockreg(x->a, MASKA, TAPA);
    x->b = clockreg(x->b, MASKB, TAPB);
    x->c = clockreg(x->c, MASKC, TAPC);
}

__device__ void
clockmaj(struct lfsrs * x) {
    uint32_t maj = majority(x);
    if(maj == ((x->a & CLKA) != 0)) {
        x->a = clockreg(x->a, MASKA, TAPA);
    }
    if(maj == ((x->b & CLKB) != 0)) {
        x->b = clockreg(x->b, MASKB, TAPB);
    }
    if(maj == ((x->c & CLKC) != 0)) {
        x->c = clockreg(x->c, MASKC, TAPC);
    }
}
```

```
__device__  uint32_t
highbit(struct lfsrs * x) {
    return ((x->a & OUTA) >> 18) ^ ((x->b & OUTB) >> 21) ^ ((x->c & OUTC) >>
       22);
}

__device__ void
run(struct lfsrs * x, uint8_t * a, uint8_t * b) {
    uint32_t lx = (threadIdx.x + (blockIdx.x*blockDim.x));
    uint32_t ix = lx * 114;
    uint32_t i, h;
    for(i = 0; i < 114; i++) {
        clockmaj(&x[lx]);
        h = highbit(&x[lx]);
        a[(i/8)+ix] |= h << (7 - (i&7));
    }
    for(i = 0; i < 114; i++) {
        clockmaj(&x[lx]);
        h = highbit(&x[lx]);
        b[(i/8)+ix] |= h << (7 - (i&7));
    }
}

__global__ void
setup(struct lfsrs * x, uint64_t * key, uint32_t * frame, uint8_t * a, uint8_t
    * b) {
    uint32_t i, t, lx;
    lx = threadIdx.x + (blockIdx.x*blockDim.x);

    for(i = 0; i < 64; i++) {
        clocklfsrs(&x[lx]);
        t = (key[lx] >> i) & 0x1;
        x[lx].a ^= t;
        x[lx].b ^= t;
        x[lx].c ^= t;
    }
    for(i = 0; i < 22; i++) {
        clocklfsrs(&x[lx]);
        t = (frame[lx] >> i) & 0x1;
        x[lx].a ^= t;
        x[lx].b ^= t;
        x[lx].c ^= t;
    }
    for(i = 0; i < 100; i++) {
        clockmaj(&x[lx]);
    }
    run(x, a, b);
}

int
main(int argc, char *argv[]) {
    struct timespec s, e, d;
    struct lfsrs *regs, *dregs;
    uint64_t *key, *dkey;
```

```
    uint32_t *frame, *dframe;
    uint8_t *a, *b, *da, *db;
    uint32_t in, inn, n;
    if(argc < 3) {
        printf("not␣enough␣args.␣cudaa5␣numblocks␣threadsperblock\n");
        return -1;
    }
    in = atoi(argv[1]);
    inn = atoi(argv[2]);
    n = in*inn;

    regs = (lfsrs *)malloc(sizeof(lfsrs)*n);
    key = (uint64_t *)malloc(sizeof(uint64_t)*n);
    frame = (uint32_t *)malloc(sizeof(uint32_t)*n);
    a = (uint8_t *)malloc(sizeof(uint32_t)*114*n);
    b = (uint8_t *)malloc(sizeof(uint32_t)*114*n);

    cudaMalloc((void **)&dregs, sizeof(lfsrs)*n);
    cudaMalloc((void **)&dkey, sizeof(uint32_t)*n);
    cudaMalloc((void **)&dframe, sizeof(uint32_t)*n);
    cudaMalloc((void **)&da, sizeof(uint32_t)*n*114);
    cudaMalloc((void **)&db, sizeof(uint32_t)*n*114);

    memset(regs, 0, sizeof(lfsrs)*n);

    key[0] = 0xefcdab8967452312;
    frame[0] = 0x134;

    dim3 dimGrid(in);
    dim3 dimBlock(inn);

    clock_gettime(CLOCK_MONOTONIC, &s);

    cudaMemcpy(dregs, regs, sizeof(lfsrs)*n, cudaMemcpyHostToDevice);
    cudaMemcpy(dkey, key, sizeof(uint32_t)*n, cudaMemcpyHostToDevice);
    cudaMemcpy(dframe, frame, sizeof(uint32_t)*n, cudaMemcpyHostToDevice);

    setup<<<dimGrid,dimBlock>>>(dregs, dkey, dframe, da, db);
    cudaDeviceSynchronize();

    cudaMemcpy(a, da, sizeof(uint32_t)*n*114, cudaMemcpyDeviceToHost);
    cudaMemcpy(b, db, sizeof(uint32_t)*n*114, cudaMemcpyDeviceToHost);

    clock_gettime(CLOCK_MONOTONIC, &e);
    d = diff(s, e);

    printf("took:␣%ld␣to␣run\n", d.tv_sec*1000000000+d.tv_nsec);

    return 0;
}
```

A.6 cudasa5.cu

```
/*
    Karl Ott
    CUDA SWAR A5/1 cipher
*/

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <cuda.h>
#include <time.h>
#include "timing.h"

/*
    indexed as lsb as 0
    a = 19 bits in length tapped at bits 13,16,17,18 clocked on bit 8
    b = 22 bits in length tapped at bits 20,21 clocked on bit 10
    c = 23 bits in length tapped at bits 7,20,21,22 clocked on bit 10
    polynomials:
        x^19 + x^18 + x^17 + x^14 + 1
        x^22 + x^21 + 1
        x^23 + x^22 + x^21 + x^8 + 1
    initalized to 0
    64 bit key is xored into the lsb of the registers
    22 bit frame is then xored into the lsb of the registers
    100 majority clocks follow with discarded output
    registers are clocked using a majority rule
    registers are now ready to produce two 114 bit sequences, down/up
    as per wikipedia: http://en.wikipedia.org/wiki/A5/1
*/

#define ALL 0xffffffff

enum {
    TAPA = 0,
    TAPB = 1,
    TAPC = 2
};

struct lfsrs {
    uint64_t a[19];
    uint64_t b[22];
    uint64_t c[23];
};

__device__ uint64_t
majority(struct lfsrs * t) {
    uint64_t i, j, k;
    i = t->a[8];
    j = t->b[10];
    k = t->c[10];
    return ((j & k) | (i & k) | (i & j));
```

```
}

__device__ void
clockreg(uint64_t * x, uint64_t t, uint64_t m) {
    uint64_t s, n, i;
    switch(t) {
    case TAPA:
        s = 18;
        n = (x[13] ^ x[16] ^ x[17] ^ x[18]);
        break;
    case TAPB:
        s = 21;
        n = (x[20] ^ x[21]);
        break;
    case TAPC:
        s = 22;
        n = (x[7] ^ x[20] ^ x[21] ^ x[22]);
        break;
    default:
        s = 0;
        m = 0;
        n = 0;
        break;
    }

    for(i = s; i > 0; i--) {
        x[i] = ((x[i] & (~m)) | (x[i-1] & m));
    }
    x[0] = (x[0] & (~m)) | (n & m);
}

__device__ void
clocklfsrs(struct lfsrs * t) {
    clockreg(t->a, TAPA, ALL);
    clockreg(t->b, TAPB, ALL);
    clockreg(t->c, TAPC, ALL);
}

__device__ void
clockmaj(struct lfsrs * t) {
    uint64_t maj = majority(t);
    clockreg(t->a, TAPA, ~(maj ^ t->a[8]));
    clockreg(t->b, TAPB, ~(maj ^ t->b[10]));
    clockreg(t->c, TAPC, ~(maj ^ t->c[10]));
}

__device__  uint64_t
highbits(struct lfsrs * t) {
    return (t->a[18] ^ t->b[21] ^ t->c[22]);
}

__device__ void
run(struct lfsrs * x, uint64_t * a, uint64_t * b) {
    uint64_t i;
```

```
    uint64_t lx = (threadIdx.x + (blockIdx.x*blockDim.x));
    uint64_t ix = (threadIdx.x + (blockIdx.x*blockDim.x)) * 114;
    struct lfsrs * t = &x[lx];
    for(i = 0; i < 114; i++) {
        clockmaj(t);
        a[i+ix] = highbits(t);
    }
    for(i = 0; i < 114; i++) {
        clockmaj(t);
        b[i+ix] = highbits(t);
    }
}

__global__ void
setup(struct lfsrs * x, uint64_t * key, uint64_t * frame, uint64_t * a,
   uint64_t * b) {
    uint64_t i, lx, kx, fx;
    lx = threadIdx.x + (blockIdx.x*blockDim.x);
    kx = (threadIdx.x + (blockIdx.x*blockDim.x))*64;
    fx = (threadIdx.x + (blockIdx.x*blockDim.x))*22;
    struct lfsrs * t = &x[lx];

    for(i = 0; i < 64; i++) {
        clocklfsrs(t);
        x[lx].a[0] ^= key[i+kx];
        x[lx].b[0] ^= key[i+kx];
        x[lx].c[0] ^= key[i+kx];
    }
    for(i = 0; i < 22; i++) {
        clocklfsrs(t);
        x[lx].a[0] ^= frame[i+fx];
        x[lx].b[0] ^= frame[i+fx];
        x[lx].c[0] ^= frame[i+fx];
    }
    for(i = 0; i < 100; i++) {
        clockmaj(t);
    }
    run(x, a, b);
}

int
main(int argc, char *argv[]) {
    struct timespec s, e, d;
    struct lfsrs * regs;
    struct lfsrs * dregs;
    uint64_t k;
    uint64_t *key, *dkey, *frame, *dframe;
    uint64_t f, i;
    uint64_t *a, *b, *da, *db;
    uint64_t in, inn;
    if(argc < 3) {
        printf("not␣enough␣args.␣cudaa5␣numblocks␣threadsperblock\n");
        return -1;
    }
```

```
    in = atoi(argv[1]);
    inn = atoi(argv[2]);
    uint64_t n = in*inn;

    regs = (lfsrs *)malloc(sizeof(lfsrs)*n);
    key = (uint64_t *)malloc(sizeof(uint64_t)*64*n);
    frame = (uint64_t *)malloc(sizeof(uint64_t)*22*n);
    a = (uint64_t *)malloc(sizeof(uint64_t)*114*n);
    b = (uint64_t *)malloc(sizeof(uint64_t)*114*n);

    cudaMalloc((void **)&dregs, sizeof(lfsrs)*n);
    cudaMalloc((void **)&dkey, sizeof(uint64_t)*n*64);
    cudaMalloc((void **)&dframe, sizeof(uint64_t)*n*22);
    cudaMalloc((void **)&da, sizeof(uint64_t)*n*114);
    cudaMalloc((void **)&db, sizeof(uint64_t)*n*114);

    memset(regs, 0, sizeof(lfsrs)*n);
    memset(key, 0, sizeof(uint64_t)*64*n);
    memset(frame, 0, sizeof(uint64_t)*22*n);

    k = 0x67452312;
    f = 0x134;
    for(i = 0; i < 64; i++) {
        key[i] = (k >> i) & 1;
    }

    for(i = 0; i < 22; i++) {
        frame[i] = (f >> i) & 1;
    }

    dim3 dimGrid(in);
    dim3 dimBlock(inn);
    clock_gettime(CLOCK_MONOTONIC, &s);

    cudaMemcpy(dregs, regs, sizeof(lfsrs)*n, cudaMemcpyHostToDevice);
    cudaMemcpy(dkey, key, sizeof(uint64_t)*n*64, cudaMemcpyHostToDevice);
    cudaMemcpy(dframe, frame, sizeof(uint64_t)*n*22, cudaMemcpyHostToDevice);

    setup<<<dimGrid,dimBlock>>>(dregs, dkey, dframe, da, db);
    cudaDeviceSynchronize();

    cudaMemcpy(a, da, sizeof(uint64_t)*n*114, cudaMemcpyDeviceToHost);
    cudaMemcpy(b, db, sizeof(uint64_t)*n*114, cudaMemcpyDeviceToHost);

    clock_gettime(CLOCK_MONOTONIC, &e);
    d = diff(s, e);

    printf("took:␣%ld␣to␣run\n", d.tv_sec*1000000000+d.tv_nsec);

    return 0;
}
```

A.7 timing.h

```
/*
    Karl Ott
    CPU timing code for use on linux systems
*/

struct timespec;

struct timespec
diff(struct timespec, struct timespec);
```

A.8 timing.c—cpp

```
/*
    Karl Ott
    CPU timing code for use on linux systems
*/

#include <time.h>
#include "timing.h"


struct timespec
diff(struct timespec start, struct timespec end)
{
    struct timespec temp;
    if ((end.tv_nsec - start.tv_nsec) < 0) {
        temp.tv_sec = end.tv_sec - start.tv_sec - 1;
        temp.tv_nsec = 1000000000L + end.tv_nsec - start.tv_nsec;
    } else {
        temp.tv_sec = end.tv_sec - start.tv_sec;
        temp.tv_nsec = end.tv_nsec - start.tv_nsec;
    }
    return temp;
}
```

A.9 CPU and GPU Specifications

Model	Clock(GHz)	# Cores	# Threads
Intel Xeon Processor E3-1275	3.40	4	8
Intel Celeron Processor 2955U	1.40	2	2
Intel Xeon Processor E3-1240 v2	3.40	4	8
GeForce GTX 670	.980	1344	N/A

Table 27: CPU and GPU specifications

A.10 Computer 1 Timings

Table 28: Computer 1 A5/1 timings

# Sims	Avg Time(us)	Std Dev(us)
1	9.36	1.54
2	16.54	2.46
4	28.76	3.94
8	56.15	8.71
16	111.68	16.00
32	221.19	37.22
64	473.64	35.74
128	916.37	91.61
256	1711.01	298.80
512	3528.13	710.20
1024	6075.73	1584.70
2048	10380.25	2868.48
4096	18119.06	3164.15
8192	30994.68	2705.22
16384	57013.67	2165.72
32768	109650.51	2360.34
65536	214626.92	1675.63

Table 29: Computer 1 SWAR A5/1 timings

# Sims	Avg Time(us)	Std Dev(us)
1	44.18	6.53
2	86.18	11.94
4	175.41	22.53
8	346.08	33.90
16	723.33	74.48
32	1432.09	179.73
64	2662.54	496.01
128	5051.27	1260.44
256	8774.52	2428.57
512	15302.88	3218.51
1024	25149.25	2501.45
2048	45236.35	1727.42
4096	87309.20	2296.51
8192	170697.73	2273.81
16384	336993.17	2084.99
32768	669431.19	2180.58
65536	1334635.33	2596.53

Table 30: Computer 1 Multithreaded A5/1 timings

# Sims	Avg Time(us)	Std Dev(us)
1	3887.04	2147.70
2	4006.64	2068.08
4	3908.64	2100.39
8	3863.39	2228.52
16	4018.53	2172.23
32	4051.28	1982.06
64	3986.52	2086.61
128	4099.53	2082.87
256	4311.73	2101.36
512	4745.66	2037.32
1024	5232.11	2049.38
2048	6729.89	2031.27
4096	9702.67	2148.20
8192	15424.30	2427.50
16384	26605.60	2675.42
32768	48658.56	4154.74
65536	93713.06	8020.75

Table 31: Computer 1 Multithreaded SWAR A5/1 timings

# Sims	Avg Time(us)	Std Dev(us)
1	3918.53	2163.97
2	3994.15	2177.49
4	3901.85	2134.79
8	3931.33	2065.11
16	4012.34	2103.15
32	4068.84	2284.55
64	4227.92	2206.63
128	4501.79	2269.21
256	5121.39	2404.48
512	7446.91	2460.33
1024	10317.88	2470.20
2048	16143.13	2970.36
4096	27004.53	3196.58
8192	48216.36	2674.46
16384	92477.43	2968.30
32768	180165.14	3266.60
65536	355111.41	3029.50

Table 32: Computer 1 CUDA A5/1 timings

# Blocks	# Threads	Avg Time(us)	Std Dev(us)
1	1	355.39	33.26
1	2	370.39	25.37
1	4	405.68	55.22
1	8	456.52	49.09
1	16	593.45	22.02
1	32	828.36	29.50
1	64	859.25	23.01
1	128	895.66	24.95
1	256	977.75	32.37
2	256	1115.04	39.71
4	256	1390.25	104.16
8	256	1983.42	157.00
16	256	2992.38	197.86
32	256	5629.98	336.15
64	256	11634.45	296.19
128	256	21676.10	369.67
256	256	41298.94	613.61

Table 33: Computer 1 CUDA SWAR A5/1 timings

# Blocks	# Threads	Avg Time(us)	Std Dev(us)
1	1	1808.12	16.26
1	2	1829.82	25.88
1	4	3558.86	17.50
1	8	6650.27	11.85
1	16	12926.45	27.81
1	32	25200.13	26.34
1	64	26279.10	33.80
1	128	26548.38	39.92
1	256	28672.33	74.37
2	256	29195.44	89.73
4	256	30560.38	131.77
8	256	43689.85	300.08
16	256	98734.94	336.78
32	256	226773.69	497.40
64	256	450144.56	641.07
128	256	844346.94	3953.57
256	256	1691433.80	5367.15

A.11 Computer 2 Timings

Table 34: Computer 2 A5/1 timings

# Sims	Avg Time(us)	Std Dev(us)
1	10.45	0.62
2	16.83	2.59
4	29.48	3.83
8	54.75	6.69
16	104.93	7.20
32	208.59	89.59
64	408.27	18.76
128	808.61	19.56
256	1608.90	20.05
512	3215.31	23.86
1024	6414.07	24.10
2048	12802.61	48.51
4096	25627.64	256.60
8192	51116.44	208.40
16384	102236.50	492.94
32768	205135.30	2578.55
65536	408824.28	1841.18

Table 35: Computer 2 SWAR A5/1 timings

1	47.56	5.37
2	86.30	8.46
4	163.43	10.11
8	324.44	17.25
16	639.61	18.67
32	1266.77	21.08
64	2531.74	25.99
128	5045.11	57.96
256	10051.13	75.46
512	20065.32	138.44
1024	40077.23	130.62
2048	80240.40	672.26
4096	160276.17	927.54
8192	320284.61	2261.59
16384	640453.53	6062.52
32768	1278926.34	8157.35
65536	2555934.66	17264.98

Table 36: Computer 2 Multithreaded A5/1 timings

1	336.44	790.65
2	319.72	698.91
4	350.19	727.83
8	406.06	732.62
16	427.54	728.78
32	475.55	729.68
64	550.37	674.65
128	840.77	756.64
256	1295.11	568.62
512	2243.28	500.59
1024	4319.15	687.34
2048	8524.32	1210.51
4096	16827.13	2284.24
8192	33470.66	4578.21
16384	66061.41	9296.40
32768	132221.03	18284.40
65536	264134.22	36523.56

Table 37: Computer 2 Multithreaded SWAR A5/1 timings

1	370.65	751.58
2	409.43	772.50
4	486.42	848.73
8	528.86	823.04
16	621.34	721.86
32	909.87	626.88
64	1450.54	435.16
128	2625.47	285.57
256	5114.09	309.67
512	10335.72	313.94
1024	20501.54	362.54
2048	40593.12	573.90
4096	80937.92	885.60
8192	161428.84	2439.10
16384	324535.79	6544.83
32768	644453.46	7615.04
65536	1285961.61	9161.41

A.12 Computer 3 timings

Table 38: Computer 3 A5/1 timings

# Sims	Avg Time(us)	Std Dev(us)
1	8.06	1.31
2	13.13	1.93
4	22.34	3.19
8	41.33	19.70
16	80.01	11.67
32	164.79	29.75
64	339.71	35.16
128	665.56	65.78
256	1283.59	202.26
512	2408.89	489.62
1024	4857.45	1111.40
2048	8106.94	2271.57
4096	13362.99	3146.31
8192	24106.57	2891.15
16384	43548.30	2715.00
32768	82167.32	2112.56
65536	161359.14	3366.82

Table 39: Computer 3 SWAR A5/1 timings

# Sims	Avg Time(us)	Std Dev(us)
1	39.93	5.76
2	73.89	8.26
4	134.74	19.47
8	295.59	31.77
16	595.06	62.92
32	1150.54	201.47
64	2162.78	432.44
128	4069.48	987.35
256	7415.01	2089.89
512	12738.21	3214.72
1024	21825.15	3029.49
2048	38568.20	2425.57
4096	73286.07	2499.72
8192	141825.17	1620.41
16384	281514.58	2320.65
32768	558285.67	5468.02
65536	1111534.60	2442.18

Table 40: Computer 3 Multithreaded A5/1 timings

# Sims	Avg Time(us)	Std Dev(us)
1	3667.01	2036.95
2	3492.65	1940.98
4	3582.20	1910.80
8	3560.61	2068.77
16	3425.35	1940.23
32	3562.53	2000.15
64	3686.91	2029.61
128	3796.42	2010.18
256	4107.76	2014.46
512	4737.11	1915.61
1024	5324.46	1929.89
2048	6451.41	1872.62
4096	8822.64	2018.84
8192	13652.14	2381.90
16384	23355.28	2476.46
32768	41915.13	3898.20
65536	79390.90	7457.43

Table 41: Computer 3 Multithreaded SWAR A5/1 timings

# Sims	Avg Time(us)	Std Dev(us)
1	3532.96	1950.60
2	3459.65	1867.79
4	3445.06	1850.67
8	3361.53	2020.45
16	3569.45	1967.82
32	3841.27	2097.76
64	4043.79	2133.23
128	4670.06	2294.20
256	5108.06	2411.75
512	6734.45	2206.95
1024	9541.64	2338.52
2048	13982.85	2793.66
4096	22625.42	3125.30
8192	38698.40	3422.34
16384	71964.14	3156.89
32768	138209.36	3412.67
65536	270804.03	3377.38